ACORNSOFT
The choice of experience in software.

Archimedes

# ANSI C

GUIDE

ACORNSOFT
The choice of experience in software.

# ANSI C

GUIDE

First published 1988
Release 2
Published by Acorn Computers Limited
Part number 0481,883

# CONTENTS

# INTRODUCTION

The Acorn C compiler for the ARM processor is a full implementation of C as defined by the October 1986 draft ANSI language standard. This manual is a reference text designed to accompany the *Draft Proposed American National Standard for Information Systems - C Programming Language* document (October 1986) and should therefore be read in conjunction with it. The compiler also has a Portable C compatibility mode.

## ABOUT THIS USER GUIDE

We assume that the reader of this User Guide has a working knowledge of C. The Guide does not act as an introduction to C. Subsequent chapters and appendices contain information on:

- Installation and use of the compiler, in *How to install and run the compiler*.
- How Acorn C implements those aspects of the language which ANSI leaves to the discretion of the implementor, in *Implementation details*.
- How Acorn C behaves in those areas covered by Appendix A.6 of the draft standard, in *Standard Implementation definition*.
- The additional libraries for the Arthur operating system and the Window Manager (Wimp), in *Arthur operating system library*.
- How to handle procedure entry and exit in assembly language, so you can write programs which interface elegantly with the code produced by the C compiler in *Assembly language interface*.
- Messages produced by the compiler, of varying degrees of seriousness, in *Errors and warnings*.
- Using the compiler in its Portable C Compiler compatibility mode, in *Using the PCC mode*.
- Calling other applications from C in a manner which allows control to return to the C program, in *Calling applications from C*.

If you need information of a more introductory nature, you could refer to the following books:

- *The C programming language* by BW Kernighan and DM Ritchie, published by Prentice-Hall, Englewood Cliffs, NJ, USA. This is the original C 'bible', still useful for a description of PCC-style C, but superseded in other respects by Harbison and Steele.

- A C *Reference Manual, 2nd. edition* by Samuel P Harbison and Guy L Steele Jr, published by Prentice-Hall, Englewood Cliffs, NJ, USA. This is a very thorough *reference* guide to C, including a useful amount of information on the proposed ANSI draft C.

- C *Language* by Friedman Wagner-Dobler, published by Pitman, 128 Long Acre, London UK.

- C *Puzzle Book* by Alan R Feuer, published by Prentice-Hall, Englewood Cliffs, NJ, USA.

- *The X/OPEN Portability Guide*, published by Elsevier Science Publishers BV, 1000 BZ, Amsterdam, Netherlands.

## CONVENTIONS USED

Throughout this User Guide, the following convention is used for text that appears on the screen, for C keywords, macros, library routines etc. A typewriter style font is used for text to be typed as is, and italics are used to denote classes of item which would be replaced in the command or message by actual objects of the appropriate type. For example:

`cc` *options filenames*

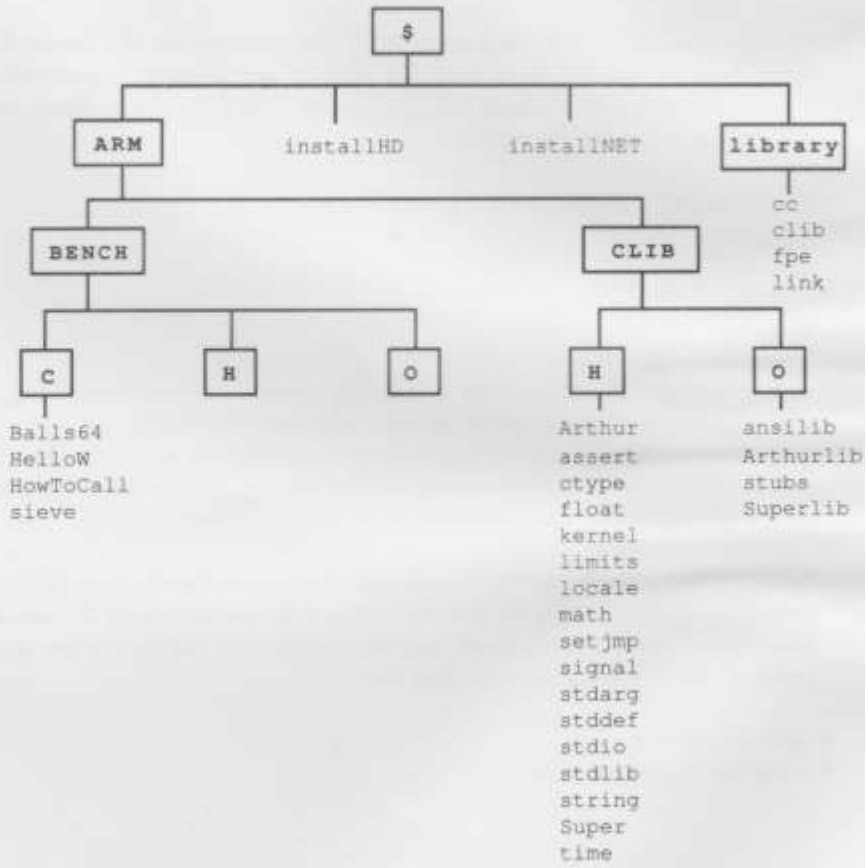
means that you type `cc` as shown, and replace *options* and *filenames* by specific examples.

# HOW TO INSTALL AND RUN THE COMPILER

This section describes, first, how to install the C compiler, second, the naming conventions used by Acorn C, and finally, how to run the compiler and control various aspects of its operation.

## INSTALLATION

The diagram below shows the directory structure of the C distribution disc.



Upon receiving your disc, you should make a working copy and keep the original in a safe place. To make the copy, enter the following command:

```
*backup 0 0 q
```

The program will prompt you to insert a destination disc (ie a blank formatted 800k disc) in place of the original disc that you wish to copy. The *Archimedes User Guide* tells you how to format a floppy disc.

The programs installHD and installNET are provided on the floppy disc to enable you to install C on a hard disc and Econet network respectively. These can be run by clicking the appropriate icon from the desktop, or from the Arthur * prompt by typing

```
*installHD
```

or

```
*installNET
```

as appropriate. If you have an Econet file server, you will need to log on with system privileges in order to copy files into the library. It is recommended that you copy the structure provided on the disc supplied, except that directories `h`, `o` and `c` will be created at user level. Remember to set file attributes for global use.

Before running the compiler, you should ensure that the floating point emulator is installed. To do this on Arthur 1.2, you must load the emulator from the C disc. If you are not sure what version of Arthur you are using, perform the following steps to find out:

- quit the desktop
- from the * prompt, type

```
*fx 0
```

A message of the form

```
Arthur 1.20 (01 September 1987) (Error number &F7)
```

will be displayed. If the version number is 1.20, insert the C disc and type

```
$.lib*.fpe
```

This installs the floating point emulator.

## GETTING STARTED

Having backed up the C distribution disc, insert your working disc in the drive and select the directory `$.ARM.BENCH` by entering the following at the * prompt:

```
*DIR $.ARM.BENCH
```

There are four example programs provided on the disc (see the directory structure diagram above):

- `Balls64` colourful graphics demonstration
- `HelloW` simple test program
- `HowToCall` see the section entitled *Calling Applications from C*
- `Sieve` standard test program

To compile and link the program `HelloW`, type

```
*cc c.hellow
```

(note that upper and lower case letters are interchangeable in the filename).

When this process is completed, the * prompt will return under a finishing message displayed on the screen. To run the program, now type

```
*hellow
```

and the program will print the mesage `Hello World` on the screen.

Balls64 makes use of some of the graphics facilities of your Archimedes personal workstation, and needs linking with the Arthur library after compilation. The compiler will do this for you if you type

```
*cc -Arthur balls64
```

and then run the program by typing

```
*balls64
```

Press the Esc key to leave this program.

The `cc` command and its associated conventions, such as where the C compiler expects to find source, library, object and image files, is covered in detail in the sections which follow.

## NAMING CONVENTIONS

The Acorn C system, in common with many other C systems, uses naming conventions to identify the four classes of file involved in the compilation and linking process. Many systems use conventional suffixes for this. For example, the suffix `.c` denotes C source files on UNIX™ and MS-DOS™ systems. This convention clashes with Acorn's use of the period character (`.`) in pathnames. It is more natural under Acorn filing systems to use a prefix mechanism, eg `c.foo`, where `c` is the directory containing C source files, and `foo` is the filename.

However, portability is an increasingly important issue in the C world. To this end, the Acorn C system recognises the 'standard' file naming conventions and performs the appropriate transformations to construct valid Arthur pathnames. The following sections summarise the conventions for referring to source, include, object and program files.

### Source files

Source files will be looked for in subdirectory `c`. To aid portability, a file `foo.c` will be looked for in `@.c.foo`, where `@` means the current directory.

### Include files

The way in which the compiler deals with included files depends on whether the name of the filename in the `#include` directive is between angled brackets < > or double quotation marks " ". The former case implies that the file is a 'system' one, the latter that it is a 'user' file. Simply put, system include files are searched for within the compiler's built-in filing system, and then in the 'system include path'. This is read from the environment variable `C$Libroot` if possible, otherwise it defaults to `$.arm.clib`. User include files – that is, those enclosed in double quotation marks – are sought in the current directory, then the system include path.

In both cases, an intermediate set of directories to search may be specified using the `-I` command-line flag. Much more information about include file processing may be found in the section on the `-I` and `-j` flags below.

### Object files

The object files created by the compiler are stored in the directory `o` within the currently selected directory. Thus the result of compiling `c.expr` will be found in `o.expr`.

### Compilation list files

If the `-list` keyword is specified, then a file containing a compilation listing for each compiled source file is created in the directory `@.l`. Thus compiling the file `c.expr` with the keyword `-list` will result in the list file `@.l.expr` being created.

### Library files

The default link step activated by the compiler searches for libraries in the 'system library path'. As with the system include path, the compiler tries to read the system library path from the environment variable `C$libroot`. If this isn't defined, the path `$.arm.clib` is used instead. In either case, `.o.` is appended to the directory name.

Standard library files are called `ansilib`, `ArthurLib` and `SuperLib`. These contain the code for the ANSI library, the Arthur-specific library and the SpringBoard/Brazil library respectively. The command line flag `-l` may be used to override this search path with one or more user-specified ones.

### Program files

The result of linking the compiled versions of the source files given in the command line with the libraries is an executable program file. This is named
`@.`*file1*, where *file1* is the name of the first source file given on the command line. This convention may be overridden by use of the `-o` flag.

As an example of these naming conventions in use, consider the directory `bench` on the distribution disc:

- The directory `bench.c` contains C program sources
- `Bench.c.sieve` is the source of a sample program which computes prime numbers by the sieve of Eratosthenes. It would be referred to in a command line as `sieve.c` or `c.sieve`.
- This compiles to produce the object file `bench.o.sieve`
- This object file is linked with the standard library to produce the executable program `bench.sieve`.

The compiler does not check whether the filenames you give are acceptable, ie contain only valid characters and are of acceptable length – this is done by the filing system.

## COMPILING A SIMPLE PROGRAM

The compiler can only be used from the * prompt. If you are using the desktop, return to the * prompt by clicking the select button on the Exit icon. Next, load the floating point emulator by typing

```
rmload $.library.fpe
```

You are now ready to run the C compiler.

The compiler is called `cc`. To run it, type:

`*cc` *options filenames*

The *options* are described in detail below. They allow you to control the compilation by, for example, overriding default names. Often, just the list of one or more filenames is all that is required following the `cc` command. For example, suppose you have created the following program:

```
#include <stdio.h>
int main()
{
        printf("Hello world!\n");
        return 0;
}
```

and stored it as `c.hello`. To compile it, you enter the command

```
*cc hello
```

Assuming that everything is correct, the compiler will give you a message similar to the following:

```
Norcroft ARM C (with debugger support) 1.65A May 20
1988
```

This compilation produces the object file `o.hello`. The compiler automatically issues the `Link` command necessary to combine the object file with the standard libraries in order to produce the executable file called `hello`. This can be executed using the command `*hello`.

You can suppress the linking stage by specifying a command line option. This, and other options, are described in the section below.

## COMPILER OPTIONS

You can control many aspects of the compiler's operation by appending options to the command `cc`. All options are prefixed by the minus sign `-`.

Options come in two forms. The first are keywords. These are multiple-character options and control Acorn or Arthur-specific aspects of the compiler. Keywords are recognised in upper or lower case. The second form of option is the flag. A flag is a single letter. The case of the letter is usually unimportant to the C compiler under discussion. However, UNIX compilers only recognise one form (either the upper or lower case one, depending on the flag) and this one should be used in preference.

The question of the case of flags is most important when you are considering porting 'make' files to other systems. By using the 'universal' syntax of the `cc` command, you can move your system to different environments with the minimum amount of work.

The keyword options are:

`-help` Give a description of the compiler's command syntax.

`-arthur` Add the Arthur interface library to the list of 'standard' libraries passed to the linker. This is only valid under the Arthur operating system.

`-super` Add the Brazil supervisor interface library to the list of 'standard' libraries passed to the linker. This is only valid under Brazil, SpringBoard and Arthur. If the `-arthur` keyword is also given under Arthur, then that takes priority.

`-pcc` Compile old-style 'portable C compiler' C. This is often referred to as Kernighan and Ritchie (or just K & R) C. This changes the syntax that is acceptable to the compiler, but the default header and library files are still used. See the section on this option for more details.

`-list` Create a listing file. This consists of lines of source interleaved with error and warning messages. Finer control over the contents of this file may be obtained using the `-f` flag (see below).

The flag options are listed below. Some of these are followed by an argument. Whenever this is the case, the compiler allows whitespace to be inserted between the flag letter and the argument. However, this is not always true of other C compilers, so the syntax given lists only the form that would be acceptable to a UNIX C compiler. Similarly, only the case of the letter that would be accepted by a UNIX C compiler is shown.

The descriptions are divided into several sections, so that flags controlling related aspects of the compiler's operation are grouped together.

**Controlling the linker**

`-c` Do not perform the Link step. This merely compiles the source program(s), leaving the object file(s) in the `o` directory.

NB This is different from the `-C` option described below.

`-l` *libs* This specifies a list of libraries which will be used in the Link command issued by the compiler. The list of libraries to be used follows the flag (with optional white-space between the flag and the list), and uses commas to separate items in the list. Alternatively, multiple `-l` flags may be used to specify more than one library. Note that the libraries given by this option are used *instead of* the standard ones, not in addition to them.

This flag is not compatible with corresponding UNIX C compiler option, which has no direct equivalent under Arthur.

### Controlling the pre-processor

The `-I` and `-j` flags control the search paths used when the compiler is looking for included files. Before describing these flags, we explain the way in which the search paths are used.

The list of paths comprises three separate elements. In order, these are the directory containing the current source file, any directories given by `-I` flags (in the order they appear on the command line), and finally the 'system include path'. In the absence of any `-j` flag on the command line, the system include path is read from the string variable `c$libroot`. In the absence of that variable, `$.arm.clib` is used.

When a directive of the form

```
#include <filename>
```

or

```
#include "filename"
```

is encountered, the compiler behaves as follows.

If *`filename`* is a 'rooted' filename, that is

- an Arthur filename beginning with a `$` or an `&`
- a UNIX filename beginning with a `/`
- an MS-DOS filename beginning with a `\`

then it is used as written (except that UNIX-style and MS-DOS-style filenames are first translated to equivalent Arthur filenames as described below).

If *`filename`* is not 'rooted' in the sense described above, then the compiler looks for it in a sequence of places (directories) called the *search path*. The search path is in three parts, in order:

- the compiler's own in-memory filing system;
- the directory containing the source file (C source or `#included` header) currently being processed by the compiler (the 'current place');
- the remainder of the path, formed by concatenating the lists of directories given as command line arguments to `-I` flags with the system search path (see below). The system search path is searched last.

  If the `#included` filename is given in angle brackets < and >, then the directory containing the current source file is not searched.

  If the `#included` filename is given in double quotation marks then the compiler's in-memory filing system is not searched.

  In both cases, to facilitate the porting of code from UNIX and MS-DOS to Arthur, UNIX-style and MS-DOS-style filenames are translated to equivalent Arthur-style filenames. For example:

```
../include/defs.h    is translated to    ^.include.h.defs
..\clx.hash.h        is translated to    ^.clx.h.hash
includes.h           is translated to    h.includes
```

  but

```
system.defs              is translated to        system.defs
```

  (Similarly, the lists of directory names given as arguments to the compiler's `-I` and `-j` command-line flags [see below] are translated to Arthur format before being used).

  When a file is found relative to an element of the search path, the name of the directory containing that file becomes the new 'current place'. When the compiler has finished processing that file it restores the old current place. So, at any given instant, there is a stack of current places corresponding to the stack of nested `#includes`.

  For example, suppose the current place is `$.include` and that the compiler is seeking the `#included` file `"sys.def.h"` (or

"sys.h.defs", "sys/defs.h", etc). Now suppose this is found in $.include.sys.h.defs. Then the new current place becomes $.include.sys and files #included by h.defs, whose names are not rooted, will be sought relative to $.include.sys.

In all this, the penultimate .c and .h components of the path are omitted. These are logically part of the filename - a filename *extension* - not logically part of the directory structure. However, directory names other than c, h, o and s are not so recognised (as filename extensions) and are used 'as is'. For example, the name sys.new.defs is exactly that: it is not translated to sys.defs.new and, if it is found, the new part of the name *does* become part of the new current path.

Initially the system path is

- the path given as an argument to the -j command-line flag (see below); *or*
- the value of the system variable c$libroot, if this is set and there is no -j flag; *otherwise*,
- $.arm.clib

In the first case, the in-memory filing system is omitted from the front of the path searched for #include <*filename*>. It can be reinstated by using the pseudo filename :mem as an argument to a -I flag or the -j flag. If :mem is included in the search path in this way, its position in the path is as specified, not necessarily first, so you can take complete control over where the compiler seeks #included files.

-I*path* This adds the specified directory to the list of places which are searched for include files (after the in-memory or source file directory, according to the type of include file). The directories are searched in the order in which they are given in -I options. The path should end with the name of a directory, with no .h., which is added automatically.

-j *dirs* This overrides the system include path with the list of directories which follows the flag. The directories are separated by commas. You can specify the memory file system in the list by using the

name `:mem` (in any case). An example is `myhdrs,:mem,$.proj.public.hdrs`.

`-j` is an Arthur-specific flag, and therefore non-portable.

`-E` If this flag is specified, only the pre-processor phase of the compiler is executed. The output from the pre-processor is sent to the standard output stream. It can be redirected to a file using the usual methods. By default, comments are stripped from the output, but see the next flag.

`-C` This is used in conjunction with `-E` above. It reinstates comments, so they appear in the output produced by the pre-processor. Note that it is different from the `-c` flag, which is used to suppress the link operation.

`-z`*mod* This flag can be used to emulate `#pragma` directives. The *mod* which follows it is the same sequence of characters that would follow the directive. See the section *`#pragma` Directives* for details.

**Controlling code generation**

The options described in this section control in some way the production of code by the compiler.

`-g`*mods* This flag is used to specify that debugging tables for use by the Arthur Symbolic Debugger should be generated. It is followed by an optional set of letters which specify the level of information required. No modifiers means 'generate all the information possible'. However, the tables can occupy large amounts of memory so it is sometimes useful to limit what is included as follows:

`-gf` Generate information on functions and top-level variables (outside functions) only.

`-gl` Generate information describing each line in the file.

`-gv` Generate information describing all variables.

The modifiers may be specified in any combination, eg `-gfv`.

`-o` *file* This flag specifies the name of a file in which the output of the linker should be stored. It overrides the default, which is to use the same name as the root name of the first source file mentioned on the command line.

`-p` This flag causes the compiler to generate code which allows 'profiling' of the program to take place. The profiling information may be printed out by calling the function `_mapstore()` just before the program terminates.

When this option is given, the compiler embeds calls to a 'count' routine at the start of each function body. Whenever it is called, the routine increments a 32-bit counter associated with the caller. The `_mapstore()` function prints the values of these counters, along with their associated addresses. It also prints the addresses of all of the program's function entry points, so that you can deduce how often each function (or part thereof) was called. Note though that this doesn't tell you how long the program spent in a routine, only how often it was there (but, usually, these are related values).

`-S` If this flag is specified, no object code is generated and, naturally, no attempt is make to link it. Instead, an assembly listing of the code produced is written to a file called `s.`*`file`*, where *`file`* is the name of the source file (stripped of any directories or suffixes).

`-D`*sym* Define *sym* as a pre-processor macro, as if by a line

```
#define sym 1
```

at the head of the source file.

`-Usym` Undefine *sym*, as if by a line

```
#undef sym
```

at the head of the source file. This may be used to cancel the effect of otherwise predefined symbols, eg `ARM`. (A macro `ARM` is predefined, and has the value 1).

**Controlling warning messages**

The `-w` option controls the suppression of warning messages. Usually the compiler is very free with its warnings, as this tends to indicate potential portability or other problems. However, too many such messages can be a nuisance in the early stages of developing a program, so they may be disabled.

`-W`*mod* If no modifier letters *mod* are given, then all warnings are suppressed. If one or more letters follow the flag, then only the class of warnings controlled by those letters are suppressed. The letters are:

`a` Give no `Use of = in a condition context` warning. This is given when the compiler encounters statements such as

```
if (a=b) {...
```

where it is quite possible that == was intended.

`c` Give no `<op> implicit cast of pointer to non-equal pointer` warning. Pointer casts should always be made explicit using `(`*type*`*)` operators, except in the case of `NULL`, which is a universal pointer.

`d` Give no `Unused static definition` warning. If an object was defined to be static and was not referenced in the same file, it can perform no useful function and is therefore redundant.

n Give no `Implicit narrowing cast` warning. This warning is issued when the compiler detects an assignment of an expression to an object of narrower width (eg `int` to `short`). This can cause problems with loss of precision for certain values.

p Allow characters after `#else` and `#endif` directives. Normally such characters are not allowed and their presence could signify trouble.

s Give no `Short is slower than int on this machine` warning. The statement is true, but there's a limit to the number of times you have to be told...

v Give no `Implicit return in non-void context` warning. This is most often caused by a return from a function which was assumed to return `int` (because no other type was specified) but is in fact being used as a `void` function.

z Give no `Array [0] found` error.

**Controlling additional compiler features**

The `-f` flag described in this section controls a variety of compiler features, including certain checks more rigorous than usual. Like the previous flag it is followed by modifier letters. At least one letter is required.

a Check for certain types of data flow anomalies. The compiler performs data flow analysis as part of code generation. The checks enabled by this option can sometimes indicate when an automatic variable has been used before it has been been assigned a value.

e Check that external names used within the file are still unique when reduced to six case-insensitive characters. Some linkers only provide six significant characters in their symbol tables. This can cause problems with clashes if a system uses two names such as

`getExpr1` and `getExpr2`, which are only unique in the eighth character. Note that the check can only be made within one compilation unit (source file) so cannot catch all such problems. (Note also that the system under discussion allows external names of up to 256 characters, so this is really a portability aid.)

`f` Do not embed function names in the code area. The compiler does this to make the output produced by the stack backtrace function (which is the default signal handler) and `_mapstore()` more readable. Removing the names from the compiler makes the code slightly smaller (typically 5%) at the expense of less meaningful backtraces and `_mapstore()` outputs.

`h` Check that all external objects are declared in some included header file, and that all static objects are used within the compilation unit in which they are defined. These checks support good modular programming practices.

`i` In the listing file (see `-list`) include the lines from any files included with directives of the form:

```
#include "file"
```

`j` As above, but for files included by lines of the form:

```
#include <file>
```

`m` Give a warning for pre-processor symbols that are defined but not used during the compilation.

`p` Report on *explicit* casts of integers into pointers, eg

```
char *cp = (char *) anInteger;
```

Implicit casts are reported anyway, unless suppressed by the `-wc` option.

u By default, the source text as 'seen' by the compiler after preprocessing (expansion) is listed. If `-fu` is specified then the *unexpanded* source text, as written by the user, is listed. Consider the line

```
p = NULL;
```

By default, this will be listed as `p=(0);`, with `-fu` specified, as `p=NULL;`.

When writing high-quality production software, you are encouraged to use at least the `-fah` options in the later stages of program devopment (the extra diagnostics produced can be annoying in the earlier stages).

## COMPILING AND LINKING

As already shown, to compile and link the simple program shown above you would type:

```
cc hello
```

This produces the executable program `hello`. To produce a program with a different name, you would use the `-o` option, eg:

```
cc -o greeting hello.c
```

This time the linker would produce a program that you could run using the command `*greeting`.

When writing programs that use several source files, you may want to compile them selectively and perform the link as a separate step. For example, if a program consists of the files `e1.c`, `e2.c` and `e3.c`, and you have just edited `e2.c`, you may want to compile this, then link the object file with the two other files:

```
cc -c e2.c
link -o expr o.e1 o.e2 o.e3 -lib $.arm.clib.o.ansilib
```

Alternatively,

```
cc -o expr e1.o e2.c e3.o
```

does the trick!

See the linker documentation for more details on linking. To maintain complex, multi-file programs, consider using a 'make' utility such as AMU, which forms part of the Software Developer's Toolbox.

To compile several different source programs and link them all together into one executable file, list all the filenames separated by spaces. The name of the executable program is taken from the first filename given unless `-o` *`progname`* is used.

For example, the command:

```
cc mainprog util extra
```

compiles the sources `c.mainprog`, `c.util` and `c.extra` into the object files `o.mainprog`, `o.util` and `o.extra`, and then links all three object files together with the standard library to produce the executable program `mainprog`.

## `#pragma` DIRECTIVES

The ANSI standard specifies a pre-processor directive `#pragma`, which provides an implementation-specific means of controlling the behaviour of the compiler.

The following `#pragma` directives are recognised by the compiler in lower or upper case. In each case *n* must be either 1 (to enable the feature) or 0 (to disable it).

**#pragma -sn**

This directive is used to enable and disable the generation of code which checks for stack overflow. If n is 1, stack checking code will be generated; this is the default action. If it is 0, no stack-checking code will be generated. If n is not present, the stack checking state is restored to its default, ie is enabled.

You should rarely need to use this option. The addition of stack-checking code to a function only adds (generally) two instructions and two cycles (less than 0.5µs) to the execution time of a function in the case of no overflow.

One occasion when it would be used is in writing a signal handler for the SIGSTAK event. When this occurs, stack overflow has already been detected, so checking for it again in the handler would result in a fatal circular recursion.

# MPLEMENTATION DETAILS

This chapter gives details of those aspects of the compiler which the draft ANSI standard identifies as implementation-defined, and some other points of interest to programmers. They are grouped here by subject; section 4 lists the points required to be documented as set out in appendix A.6 of the draft standard.

## IDENTIFIERS

Identifiers can be of any length. They are truncated by the compiler to 256 characters, all of which are significant (the standard requires a minimum of 31).

The source character set is determined by the host operating system. Upper and lower case characters are distinct in all identifiers, both internal and external.

## DATA ELEMENTS

The sizes of data elements are as follows:

| Type | Size in bits | |
|---|---|---|
| `char` | 8 | |
| `short` | 16 | |
| `int` | 32 | |
| `long` | 32 | |
| `float` | 32 | |
| `double` | 64 | |
| `long double` | 64 | (subject to future change) |
| `all pointers` | 32 | |

Integers are represented in two's complement form.

Data items of type `char` are `unsigned` by default, though they may be explicitly declared as `signed char` or `unsigned char`. Single-character constants are thus always positive.

Floating point quantities are stored in the IEEE format. In double and long double quantities, the word containing the sign, the exponent and the most significant part of the mantissa is stored at the lower machine address.

### Limits: `h.float` and `h.limits`

The standard defines two headers, `h.limits` and `h.float`, which contain constant declarations describing the ranges of values which can be represented by the arithmetic types. The standard also defines minimum values for many of these constants.

The following table sets out the values in these two headers on the ARM, and a brief description of their significance. See the draft standard for a full definition of their meanings.

Number of bits in smallest object that is not a bit field (ie a byte):

```
CHAR_BIT                8
```

Numeric ranges of integer types: The column on the left gives the numerical values. The column on the right gives the bit patterns (in hexadecimal) that would be interpreted as these values in C. Note though that when entering constants you must be careful about the size and signed-ness of the quantity. Furthermore, constants are interpreted differently in decimal and hexadecimal/octal. See the ANSI standard or *Harbison and Steele* for more details.

```
CHAR_MAX              255             0xff
CHAR_MIN                0             0x00

SCHAR_MAX             127             0x7f
SCHAR_MIN            -128             0x80
UCHAR_MAX             255             0xff

SHRT_MAX            32767           0x7fff
SHRT_MIN           -32768           0x8000
```

```
USHRT_MAX          65535          0xffff

INT_MAX       2147483647     0x7fffffff
INT_MIN      -2147483648     0x80000000
UINT_MAX      4294967295     0xffffffff

LONG_MAX      2147483647     0x7fffffff
LONG_MIN     -2147483648     0x80000000
ULONG_MAX     4294967295     0xffffffff
```

Characteristics of floating point:

```
FLT_RADIX          2
FLT_ROUNDS         1
```

Ranges of floating types:

```
FLT_MAX            3.40282347e+38F
DBL_MAX            1.79769313486231571e+308
LDBL_MAX           1.79769313486231571e+308

FLT_MIN            1.17549435e-38F
DBL_MIN            2.22507385850720138e-308
LDBL_MIN           2.22507385850720138e-308
```

Ranges of base two exponents:

```
FLT_MAX_EXP          128
DBL_MAX_EXP         1024
LDBL_MAX_EXP        1024

FLT_MIN_EXP        (-125)
DBL_MIN_EXP       (-1021)
LDBL_MIN_EXP      (-1021)
```

Ranges of base ten exponents:

```
FLT_MAX_10_EXP        38
DBL_MAX_10_EXP       308
LDBL_MAX_10_EXP      308

FLT_MIN_10_EXP     (-37)
DBL_MIN_10_EXP    (-307)
LDBL_MIN_10_EXP   (-307)
```

Decimal digits of precision:

```
FLT_DIG                6
DBL_DIG               15
LDBL_DIG              15
```

Digits (base two) in mantissa:

```
FLT_MANT_DIG          24
DBL_MANT_DIG          53
LDBL_MANT_DIG         53
```

Smallest positive values such that (1.0 + x != 1.0):

```
FLT_EPSILON       1.19209290e-7F
DBL_EPSILON       2.2204460492503131e-16
LDBL_EPSILON      2.2204460492503131e-16L
```

## STRUCTURED DATA TYPES

The draft standard leaves details of the layout of the components of structured data types up to each implementation. The following points apply to the Acorn C compiler:

- Structures are aligned on word boundaries.
- Structures are arranged with the first-named component at the lowest address.

- `char` components are placed in adjacent bytes.
- `short` components are aligned at even-addressed bytes.
- All other arithmetic type components are word-aligned, as are pointers and `int`s containing bitfields.
- The only valid type for bitfields is `int`, either signed or unsigned.
- A bitfield of type `int` is treated as unsigned by default.
- Bitfields must be contained within the 32 bits of an `int`.
- Bitfields are allocated within `int`s so that the first field specified occupies the least significant bits of the word.

## POINTERS

The following points apply to pointer types:

- Adjacent bytes have addresses which differ by one.
- The macro `NULL` expands to the value 0, with a pointer type.
- Casting between integers and pointers results in no change of representation.
- The compiler faults casts between pointers to functions and pointers to data.

**Pointer subtraction**

When two pointers are subtracted, the difference is obtained as if by the expression:

`((int)a - (int)b) / (int)sizeof` (*type pointed to*)

If the pointers point to objects whose size is no greater than four bytes, word alignment of data ensures that the division will be exact in all cases. For longer types, such as doubles and structures, the division may not be exact

unless both pointers are to elements of the same array. Moreover the quotient may be rounded up or down at different times, leading to potential inconsistencies.

## ARITHMETIC OPERATIONS

The compiler performs all of the 'usual arithmetic conversions' set out in the draft standard.

The following points apply to operations on the integral types:

- All signed integer arithmetic uses a two's complement representation.
- Bitwise operations on signed integral types follow the rules which arise naturally from two's complement representation.
- Right shifts on signed quantities are arithmetic.
- Any quantity which specifies the amount of a shift is treated as an unsigned 8-bit value.
- Any value to be shifted is treated as a 32-bit value.
- Left shifts of more than 31 give a result of zero.
- Right shifts of more than 31 give a result of zero from an unsigned or positive signed value, −1 from a negative signed value.
- The remainder on integer division has the same sign as the divisor.
- If a value of integral type is truncated to a shorter signed integral type, the result is obtained by masking the original value to the length of the destination and then sign extending.
- Conversions between integral types never cause exceptions to be raised.
- Integer overflow does not cause an exception to be raised.

- Integer division by zero causes an exception to be raised.

The following points apply to operations on floating types:

- The ARM's floating point registers are wider than stored floating point numbers, so that some values may be computed to a slightly higher precision than the stated limits imply.
- When a `double` or `long double` is converted to a `float`, rounding is to the nearest representable value.
- Conversions from floating to integral types cause exceptions to be raised only if the value cannot be represented in a `long int` (or `unsigned long int` in the case of conversion to an `unsigned int` type).
- Floating point underflow is not detected; any operation which underflows returns zero.
- Floating point overflow causes an exception to be raised.
- Floating point divide by zero causes an exception to be raised.

## EXPRESSION EVALUATION

The compiler performs the 'usual arithmetic conversions' (promotions) set out in the draft standard before evaluating any expression.

- The compiler may re-order expressions involving only associative and commutative operators, even in the presence of parentheses.
- Between sequence points, the compiler may evaluate expressions in any order, regardless of parentheses. Thus the side effects of expressions between sequence points may occur in any order.

- Similarly, the compiler may evaluate function arguments in any order; moreover, this order may change from release to release.

- The unary + operator, as defined in the draft standard, gives a means of specifying a sequence point other than the defined sequence points.

## IMPLEMENTATION LIMITS

The draft standard sets out certain minimum 'translation limits' which a conforming compiler must cope with; you should be aware of these if you are porting applications to other compilers. A summary is given here. The 'mem' limit indicates that no limit is imposed other than that of available memory.

| Description | Requirement | Archimedes C |
|---|---|---|
| Nesting levels of compound statements and iteration/selection control structures | 15 | mem |
| Nesting levels of conditional compilation | 6 | mem |
| Declarators modifying a basic type | 12 | mem |
| Expressions nested by parentheses | 127 | mem |
| Significant characters | | |
| – in internal identifiers and macro names | 31 | 256 |
| – in external identifiers | 6 | 256 |
| External identifiers in one source file | 511 | mem |
| Identifiers with block scope in one block | 127 | mem |
| Macro identifiers in one source file | 1024 | mem |
| Parameters in one function definition/call | 31 | 50 |
| Parameters in one macro definition/invocation | 31 | mem |
| Characters in one logical source line | 509 | no limit |
| Characters in a string literal | 509 | mem |
| Bytes in a single object | 32767 | mem |
| Nesting levels for #included files | 8 | mem |
| Case labels in a `switch` statement | 255 | mem |
| `atexit`-registered functions | 32 | 33 |

# STANDARD IMPLEMENTATION DEFINITION

This chapter discusses aspects of the compiler which aren't defined by the ANSI draft standard, but are implementation-defined and must be documented.

Appendix A.6 of the draft standard collects together information about portability issues; section A.6.3 lists those points which are implementation defined, and directs that each implementation shall document its behaviour in each of the areas listed. This chapter corresponds to appendix A.6.3, answering the points listed in the appendix, under the same headings and in the same order.

## ENVIRONMENT (A.6.3.1)

- Arguments to `main()` are the words of the command line, delimited by spaces.

- The standard input, output and error streams, `stdin`, `stdout`, and `stderr` can be redirected at runtime in the following way. For example, if `copy` is a compiled and linked program which simply copies the standard input to the standard output, the following line:

  ```
  *copy { < infile }
  ```

  runs the program, redirecting `stdin` to the file `infile`. Note that

  ```
  *copy { > outfile }
  ```

  redirects both `stdout` and `stderr` to the file `outfile`; it is not possible to redirect `stderr` alone.

- Command-line arguments redirecting `stdin` or `stdout` do not appear in the list of arguments to `main()`.

- For full details of redirection in Arthur, see the *Archimedes Programmer's Reference Manual*, volume 1.

## IDENTIFIERS (A.6.3.2)

- 256 characters are significant in identifiers without external linkage. (Allowed characters are letters, digits, and underscores.)
- 256 characters are significant in identifiers with external linkage. (Allowed characters are letters, digits, and underscores.)
- Case distinctions are significant in identifiers with external linkage.

## CHARACTERS(A.6.3.3)

The characters in the source character set are to ISO 8859-1 (Latin Alphabet), a superset of the standard ASCII character set. The printable characters are those in the range 32 to 126 and 160 to 255. All printable characters may appear in string or character constants, and in comments.

- The execution character set is identical to the source character set.
- There are four `chars` in an `int`. The bytes are ordered from least significant at the lowest address to most significant at the highest address.
- There are eight bits in a character in the execution character set. The bits are ordered such that the leftmost bit is most significant.
- Characters of the source character set in string literals and character constants map identically into characters in the execution character set.
- A character constant containing more than one character has the type `int`. Up to four characters of the constant are represented in the integer value. The first character contained in the constant occupies the lowest-addressed byte of the integer value; up to three following characters are placed at ascending addresses. Unused bytes are filled with the `NUL` character. This is not portable.
- A 'plain' `char` is treated as unsigned.

- Escape codes are:

| Escape sequence | Char value | Description |
|---|---|---|
| \a | 7 | Attention (bell) |
| \b | 8 | Backspace |
| \f | 12 | Form feed |
| \n | 10 | Newline |
| \r | 13 | Carriage return |
| \t | 9 | Tab |
| \v | 11 | Vertical tab |
| \xnn | *nn* | ASCII code in hexadecimal |
| \nnn | *nnn* | ASCII code in octal |

## INTEGERS(A.6.3.4 )

The representations and sets of values of the integral types have been set out above in *Implementation details*, *Data elements*. Note also that:

- The result of converting an integer to a shorter signed integer, if the value cannot be represented, is as if the bits in the original value which cannot be represented in the final value were masked out, and the resulting integer sign-extended. The same applies when you convert an unsigned integer to a signed integer of equal length.

- Bitwise operations on signed integers yield the expected result given two's complement representation. No sign extension takes place.

- The sign of the remainder on integer division is the same as defined for the function `div()`.

- Right shift operations on signed integral types are arithmetic.

## FLOATING POINT (A.6.3.5)

The representations and ranges of values of the floating point types have been given above in *Implementation details*, *Data elements*. Note also that:

- When a floating point number is converted to a shorter floating point one, it is rounded to the nearest representable number.

- The properties of floating point arithmetic accord with IEEE 754.

## ARRAYS AND POINTERS (A.6.3.6)

The ANSI draft standard specifies three areas in which the behaviour of arrays and pointers must be documented. The points to note are:

- The type `size_t` is defined as `unsigned int`.

- Casting pointers to integers and vice-versa involves no change of representation. Thus any integer obtained by casting from a pointer will be positive.

- The type `ptrdiff_t` is defined as `(signed) int`.

## REGISTERS (A.6.3.7)

In the Acorn C compiler, you can declare up to six objects as having the storage class `register`. The valid types are:

- any integer type

- any pointer type

- any structure type which contains only bitfields and which is no more than one word long.

Note that other variables, not declared as `register`, may be held in registers for extended periods, and that `register` variables may be held in memory for some periods.

## STRUCTURES, UNIONS AND BIT-FIELDS(A.6.3.8)

The Acorn C compiler handles structures in the following way:

- When a member of a union is accessed using a member of a different type, the resulting value can be predicted from the representation of the original type. No error is given.
- Structures are aligned on word boundaries. Characters are aligned in bytes, shorts on even numbered byte boundaries and all other types, except bitfields, are aligned on word boundaries. Bitfields are parts of `int`s, themselves aligned on word boundaries.
- A 'plain' bitfield (declared as `int`) is treated as `unsigned int`.
- A bitfield which does not fit into the space remaining in an `int` is placed in the next `int`.
- The order of allocation of bitfields within `int`s is such that the first field specified occupies the least significant bits of the word.
- Bit fields do not straddle storage unit (`int`) boundaries.

## DECLARATORS (A.6.3.9)

The number of declarators that may modify a basic type is limited only by available memory.

## STATEMENTS(A.6.3.10)

The number of `case` values in a `switch` statement is limited only by memory.

## PRE-PROCESSING DIRECTIVES (A.6.3.11)

- A single-character character constant in a pre-processor directive cannot have a negative value.

- The standard header files are contained within the compiler itself. The mechanism for translating the standard suffix mechanism to an infix notation is described in *Running the Compiler, Naming conventions*.

- Quoted names for includable source files are supported. The rules for directory searching are given in *Running the Compiler, Controlling the pre-processor*.

- The recognized `#pragma` directives and their meaning are described in *#Pragma directives*.

## LIBRARY FUNCTIONS (A.6.3.12)

When using library functions in the Acorn C compiler, note the following points:

- .The macro `NULL` expands to the integer constant 0.

- If a program redefines a reserved external identifier, then an error may occur when the program is linked with the standard libraries. If it is not linked with standard libraries, no error will be detected.

- The `assert()` function prints the following message:

  `*** assertion failed:` *expression*`, file` *filename*`, line` *line-number*

  and then calls the function `abort()`.

- The functions:

```
isalnum()
isalpha()
iscntrl()
islower()
isprint()
isupper()
ispunct()
```

usually test only for characters whose values are in the range 0 to 127 (inclusive). Characters with values greater than 127 return a result of 0 for all of these functions, except `iscntrl()` which returns non-zero for 0 to 32 (excluding `\n`, `\r`, `\v` and `\t`) and 128 to 255.

After the call `setlocale(LC_CTYPE,"ISO8859-1")` the following rules apply for characters:

0 to 31 except `\n`, `\r`, `\v` and `\t` are control characters
128 to 159 are control characters
192 to 223 except 215 are upper case
224 to 255 except 247 are lower case
160 to 191, and 215 and 247 are punctuation

The results returned by the functions reflect this.

- The mathematical functions return the following values on domain errors:

| Function | Condition | Returned value |
|---|---|---|
| `log(x)` | `x <= 0` | `-HUGE_VAL` |
| `log10(x)` | `x <= 0` | `-HUGE_VAL` |
| `sqrt(x)` | `x < 0` | `-HUGE_VAL` |
| `atan2(x,y)` | `x = y = 0` | `0.0` |
| `asin(x)` | `abs(x) >= 1` | `P1/2` |
| `acos(x)` | `abs(x) >= 1` | `0.0` |

Where `-HUGE_VAL` is written above, a number is returned which is defined in the header `h.math`. Consult the `errno` variable for the error number.

- The mathematical functions do not set `errno` on underflow range errors.
- The `setjmp()` function may be called in any expression context..
- The set of signals for the `signal()` function is as follows:

```
SIGABRT     Abort
SIGFPE      Arithmetic exception
SIGILL      Illegal instruction
SIGINT      Attention request from user
SIGSEGV     Bad memory access
SIGTERM     Termination request
SIGSTAK     Stack overflow
```

- The default handling of all the signals recognised is to print a suitable message followed by a stack backtrace. This default behaviour applies at program startup.
- The last line of a text stream does not require a terminating newline character.
- Text lines consisting solely of space characters followed by a newline character are read back exactly as written.
- No `NUL` characters are appended to a binary output stream.
- File buffering is performed as defined in the draft standard (section 3.9.3).
- A zero-length file *does* exist.
- The validity of file names is defined by the host computer's filing system.
- The same file can be open many times for reading, and once for writing or updating. A file cannot however be open for reading on one stream and for writing or updating on another.

- Note also the following points about library functions:

`remove()` Cannot remove an open file.

`rename()` The effect of calling the `rename()` function when the new name already exists is dependent on the host filing system. Not all renames are valid: examples of invalid renames include `("net:file1","net:$.file2")` and `("net:file1","adfs:file2")`.

`fprintf()` Prints `%p` arguments in hexadecimal format as if a precision of 8 had been specified. If the variant form is selected, the number is preceded by the character `@`.

`fscanf()` Treats `%p` arguments identically to `%x` arguments.

`fscanf()` Always treats the character `-` in a `%[` argument as a literal character.

`ftell()` Never reports failure.

`perror()` Generates the following messages:

| Error: | Message: |
|---|---|
| 0 | `No error (errno = 0)` |
| EDOM | `EDOM - function argument out of range` |
| ERANGE | `ERANGE - function result not representable` |
| ESIGNUM | `ESIGNUM - illegal signal number to signal() or raise()` |
| others | `Error code` *number* `has no associated message` |

`abort()` Closes all open files, and deletes all temporary files.

`getenv()` Returns the value of the named Arthur Environmental variable, or NULL if the variable had no value.
eg
```
root = getenv ("C$libroot");
  if (root == NULL) root = "$.arm.clib";
```

`system()` See the chapter *Calling other programs from C* for a discussion of the use of this function.

`strcmp()` etc. The value returned by the memory- and string-comparison functions is in the range -255 to +255. The value is negative if and only if the first string is lexically less than the second. For example, suppose S1 and S2 differ first in the third character. Then strcmp (S1,S2)<0 iff S1[2]<s2[2].

`strerror()` The error messages given by this function are identical to those given by the `perror()` function.

`clock()` Returns the time taken by the program since its invocation, as indicated by the host's operating system.

`time()` Returns the date/time in seconds past 1-Jan-1970.

`setlocale()` Only has the standard `LC_` macros defined for it. The function call `setlocale(LC_CTYPE,"ISO8859-1")` alters the behaviour of the `ctype.h` functions, as described in the section *CHARACTERS (A.6.3.3)* above.

Local time zones and Daylight Saving Time are not implemented. The values returned will always indicate that the information is not available.

# ARTHUR OPERATING SYSTEM LIBRARY

To use Arthurlib functions, their declarations must be inserted in the user's code by means of a `#include` line. As an example, here is the 'hello world' program again, here using the `mode()` function to change screen mode:

```
#include <stdio.h>
#include <Arthur.h>
int main()
{
        mode(7);
        printf("Hello world!\n");
        return 0;
}
```

When the above program is compiled and linked, the `-arthur` option has to be used:

```
cc -arthur hello
```

This causes the linker to use the `arthurlib` library in addition to the usual `ansilib` one.

Because it is quite possible for the names of the functions and variables declared in `h.arthur` to clash with other identifiers used in a program, a facility exists to enable an alternative set of names to be used. These names are longer, but are less likely to clash.

If the macro symbol `ARTHUR_NEW_NAMES` is defined before the Arthur header file is included, then all the names documented in this chapter must be prefixed by `art_` before they can be used. Here is yet another version of the hello program using this method:

```
#define ARTHUR_NEW_NAMES
#include <stdio.h>
#include <Arthur.h>
int main()
{
        art_mode(7);
        printf("Hello world!\n");
        return 0;
}
```

An alternative way of achieving the same effect as the `#define` line above would be to use the command line option `-DARTHUR_NEW_NAMES`.

## GENERAL ARTHURLIB FUNCTIONS

These functions deal with general I/O features of Arthur, including graphics, sound and keyboard. In general their functionality emulates that of similarly named BASIC keywords. Brief descriptions are given below, but you are recommended to refer to the BASIC keyword section of the User Guide for comprehensive descriptions.

Functions such as `osfile()` are essentially those described in detail in the *Arthur Programmer's Reference Manual*. Any C structures referred to are defined in the Arthur header file `<Arthur.h>` (ie `$.arm.clib.h.Arthur`). In the function declarations, the ANSI prototype facility to give names as well as types to arguments is used. This makes the arguments' use a little more self-explanatory.

### Screen calls

`void circle(int x,int y,int rad);` – Draws a circular outline with centre at co-ordinates (x,y) with radius `rad`. Equivalent to SWI `OS_Plot` with R0=4, R1=x, R2=y followed by a SWI `OS_Plot` with R0=0x95, R1=`rad`+x and R2=y

`void circlefill(int x,int y,int rad);` – Draws a solid circle:, arguments as above. Equivalents as above but R0=0x9D on the second call.

`void clg(void);` – Clears graphics window to the graphics background colour. Equivalent to SWI `OS_WriteI+0x10`.

`void cls(void);` – Clears text window to the text background colour. Equivalent to SWI `OS_WriteI+0x0c`.

`void colour(int col);` – Sets text foreground/background colour to `col`. Equivalent to SWI `OS_WriteI+0x11` followed by SWI `OS_WriteC` with R0=`col`.

`void cursor(int type);` – Alters cursor appearance. The argument has the following meanings:

| Type | Meaning |
|---|---|
| 0 | Hide cursor |
| 1 | Show cursor |
| 2 | Steady cursor |
| 3 | Flashing cursor |

Equivalent to the `OS_WriteC` sequence 0x17,1,`type`,0,0,0,0,0,0,0.

`void draw(int x,int y);` – Draws a line to the specified absolute coordinates. Equivalent to SWI `OS_Plot` with R0=5, R1=x and R2=y.

`void drawby(int dx,int dy);` – Draws a line to coordinates specified relative to current graphics cursor. Equivalent to SWI `OS_Plot` with R0=1, R1=dx and R2=dy.

`void fill(int x,int y);` – Flood-fill an area in the current foreground colour, starting from (x,y). Equivalent to SWI `OS_Plot` with R0=0x85, R1=x and R2=y.

`void gcol(int action,int col);` – Set action and colour for graphics foreground or background plotting. Equivalent to a SWI `OS+WriteI+0x12`

followed by two OS_WriteCs with R0=action and R0=col respectively.

void gwindow(int x1,int y1,int x2,int y2); – Set up a graphics window. The bottom left and top right corners of the window have the co-ordinates (x1,y1) and (x2,y2) respectively. Equivalent to the OS_WriteC sequence 0x18,x1;y1;x2;y2; where a semi-colon implies the value is sent as two bytes, low byte first.

void mode(int mde); – Set screen mode to mde. Equivalent to a SWI OS_WriteI+0x16 followed by a SWI OS_WriteC with R0=mde.

void move(int x,int y); – Move graphics cursor to absolute position (x,y). Equivalent to SWI OS_Plot with R0=4, R1=x and R2=y.

void moveby (int dx,int dy); – Move graphics cursor to a position relative to its current position. Equivalent to SWI OS_Plot with R0=0, R1=dx and R2=dy.

void origin(int x,int y); – Moves the graphics origin to absolute coordinates given. Equivalent to the OS_WriteC sequence 29,x;y;.

void palette(int l,int p,int r,int g,int b); – Physical to logical colour definition. The arguments correspond to the five bytes which follow an OS_WriteC with R0=0x13, ie logical colour, physical colour, red component, green component and blue component. See the description of VDU 19 in the *User Guide* for more details.

void plot(int type,int x,int y); – Perform an operating system plot operation. This is equivalent to a SWI OS_Plot with R0=type, R1=x and R2=y.

int point(int x,int y); – Find the logical colour of the graphics pixel indicated by coordinate arguments. Equivalent to a SWI OS_ReadPoint with R0=x, R1=y and returning the value in R2 on exit.

int pos(void); – Return the x-coordinate of the text cursor. Equivalent to a SWI OS_Byte with R0=0x86 and returning the value in R1 on exit.

`void rectangle(int x,int y,int width,int height);` – Plot a rectangular outline. The first two arguments give the co-ordinates of the bottom left corner (for positive `width` and `height`). Equivalent to the following sequence of `OS_Plots`:

| R0 | R1 | R2 |
|---|---|---|
| 4 (Move abs) | x | y |
| 1 (Line rel) | width | 0 |
| 1 (Line rel) | 0 | height |
| 1 (Line rel) | -width | 0 |
| 1 (Line rel) | 0 | -height |

`void rectanglefill(int x,int y,int width,int height);` – Plot a solid rectangle. Equivalent to a SWI `OS_Plot` with R0=4, R1=x, R2=y followed by a SWI `OS_Plot` with R0=0x61, R1=`width` and R2=`height`.

`void stringprint(char *str);` – Prints the null-terminated string pointed to by `str` on the currently enabled (Arthur) output streams. Equivalent to a SWI `OS_Write0` with R0=`str`.

`void tab(int x,int y);` – Position text cursor at (x,y) relative to the text window origin. Equivalent to a SWI `OS_WriteI+0x1f` followed by two calls to `OS_WriteC` with R0=x then y.

`void tint(int type,int tnt);` – Set grey level of a colour. The first parameter gives the colour whose tint is to be set, as follows:

| Type | Colour |
|---|---|
| 0 | Text foreground |
| 1 | Text background |
| 2 | Graphics foreground |
| 3 | Graphics background |

Equivalent to the `OS_WriteC` sequence 0x17,0x11,`type`,`tnt`,0,0,0,0,0,0.

`void vdu(int ch);` – Sends `ch` to the current VDU streams. Equivalent to a SWI `OS_WriteC` with R0=ch.

`void vduw(int ch2);` – Sends `ch2` as two characters to the current VDU streams in the order `ch2 & 0xff, (ch2 & 0xff00) >> 8`. Equivalent to two calls to SWI `OS_WriteC`.

`void vduq(int n,...);` – Sends `n` characters to the current VDU streams. `n` integer arguments follow the first argument, and each of these is interpreted as a VDU byte. Equivalent to `n` calls to SWI `OS_WriteC`.

`int vpos(void);` – Return the current text cursor y co-ordinate.

**Keyboard/mouse calls**

`int get(void);` – Return a character code from the currently selected input stream. Equivalent to a SWI `OS_ReadC` where R0 is returned as the result.

`int inkey(int delay);` – Return a character code from the input stream or keyboard, with timing features (as BASIC). Equivalent to a SWI `OS_Byte` with R0=0x81, R1=`delay % 0x100` and R2=`delay/0x100`. The return value is derived from R1 and R2 on exit.

```
int mouseX(void);
int mouseY(void);
int mouseB(void);
```

These three mouse functions return the position and button status of the mouse. Note that they should not be used with the WIMP functions described in the next section – the WIMP system provides special mouse functions.

**Sound calls**

`int get_beat(void);` – Read current beat value. Equivalent to a SWI `Sound_QBeat` with R0=0 on entry and using R0 on exit as the return value.

`int get_beats(void);` – Read beat counter cycle length. Equivalent to a SWI `Sound_QBeat` with R0=-18 on entry and using R0 on exit as the return value.

`int get_tempo(void);` – Read rate at which beat counter counts. Equivalent to a SWI `Sound_QTempo` with R0=0 on entry and using R0 on exit as the return value.

`void set_beats(int beats);` – Set beat counter cycle length to `beats`, so that it cycles from 0..`beats-1`. Equivalent to a SWI `Sound_QBeat` with R0=`beats`.

`void set_tempo(int tempo);` – Set rate at which beat counter counts to `tempo`. Equivalent to a SWI `Sound_QTempo` with R0=`tempo`.

`void sound(int chan, int amp, int pitch, int dur, int when);` Make or schedule a sound. Arguments as BASIC, except that `when = -2` implies an unsynchronised sound event . If `when!=-2` then it is equivalent to a SWI `Sound_QSchedule` with R0=`when`, R1=0 and R2/R3 containing packed versions of the first four arguments. If `when = -2` then it is equivalent to a SWI `Sound_Control` with R0..R3 set to the first four arguments,

`.void sound_off(void);` – Deactivate sound system. Equivalent to a SWI `Sound_Enable` with R0=1.

`.void sound_on(void);` – Activate sound system. Equivalent to a SWI `Sound_Enable` with R0=2.

`void stereo(int chan, int pos);` – Set stereo position for specified channel. Equivalent to a SWI `Sound_Stereo` with R0=`chan`, R1=`pos`.

`void voices(int vces);` – Set number of active sound channels to `vces` (1, 2, 4 or 8 ). Equivalent to a SWI `Sound_Configure` with R0=`vces` and R1..R4=0.

**Miscellaneous calls**

`int adval(int chan);` – Reads data from an analogue port or gives buffer data. Equivalent to a SWI `OS_Byte` with R0=0x80 and R1=`chan`.

The six functions below provide access to the obvious operating system routines indicated by their names. For details of C structures involved, see the `h.Arthur` header file. Error returns are as for the `swix()` function – see below.

`error *osargs(reg_set *regs);` – Perform a SWI `OS_Args`. The arguments to the call are found in the `reg_set` struct whose address is passed as an argument. Only R0..R2 are used/altered by the call. If there was no error, `NULL` is returned, otherwise a pointer to an `error` structure is returned.

`error *osbyte(reg_set *regs);` – Perform a SWI `OS_Byte`. The arguments to the call are found in the `reg_set` struct whose address is passed as an argument. Only R0..R2 are used/altered by the call. If there was no error, `NULL` is returned, otherwise a pointer to an `error` structure is returned.

`error *osfile(osfile_block *pb);` – Perform a SWI `OS_File`. The arguments to the call are found in the `osfile_block` struct whose address is passed as an argument. If there was no error, `NULL` is returned, otherwise a pointer to an `error` structure is returned.

`error *osfind(reg_set *regs);` – Perform a SWI `OS_Find`. The arguments to the call are found in the `reg_set` struct whose address is passed as an argument. Only R0..R2 are used/altered by the call. If there was no error, `NULL` is returned, otherwise a pointer to an `error` structure is returned.

`error *osgbpb(osgbpb_block *pb);` – Perform a SWI `OS_GBPB`. The arguments to the call are found in the `osgbpb_block` struct whose address is passed as an argument. If there was no error, `NULL` is returned, otherwise a pointer to an `error` structure is returned.

`error *osword(int type,void *pb);` – Perform a SWI `OS_Word`. The `type` argument gives the `OS_Word` number, and the `pb` argument points to the free-format `OS_Word` parameter block. If there was no error, `NULL` is returned, otherwise a pointer to an `error` structure is returned.

`unsigned rnd(unsigned seed);` – Return a BASIC-type random number.

`reg_set swi(int swiNum, reg_set *regs);` – This allows general access to operating system SWI routines. This function does not trap any errors. It is passed arguments to indicate an action number and a pointer to an input 'register set' (see `h.Arthur`). It returns an output 'register set'. Equivalent to a SWI `swiNum` with R0..R9 set up from `*regs` and returning R0..R9 on exit.

Note that the input register set is passed by address whereas the result is returned as a structure value. Thus a typical call would look like this:

```
myRegs = swi(OS_SWINumberFromString, &myRegs);
```

Note also that if an error occurs, it is not reported and there is no way to detect it. It is therefore only sensible to use this call for SWIs which will 'never' generate errors.

`error *swix(int swiNum, reg_set *regs);` – This allows general access to operating system SWI routines, as the previous one does. The arguments are the same as `swi()`. However, the 'X' form of SWI `swiNum` is used (by setting bit 17), so that errors may be reported. If the SWI executes without an error, `NULL` is returned. If an error does occur, a pointer to an `error` structure is returned (see `Arthur.h` for its fields). In

either case, `*regs` is updated with the return values of R0..R9. (So in the case of an error, `regs->r[0]` will also contain the error pointer.)

## ARTHUR WIMP FUNCTIONS

Applications working under Arthur are provided with a comprehensive system of WIMP facilities. To use these, the relocatable modules `WindowManager` and `FontManager` must be present (either resident in ROM or loaded into the relocatable module area).

Writing applications to use the WIMP system is straightforward, provided you have read the *Window manager* chapter of the *Programmer's Reference Manual*. You are also recommended to refer to the *Archimedes Reference Guide*. The C interfaces to the WIMP functions are, therefore, listed here with only brief descriptions.

The structures referred to in this section are defined in `<Arthur.h>`. The functions are listed below in order of the corresponding `Wimp` SWI number.

All of the functions have an `error *` argument to show where to put error data if an error is generated. If there is no error, the `errnum` field of the `error` structure will be zero on return.

`int w_initialise(error *);` – Close and delete all windows; returns wimp version number.

`int create_wind(wind_block *, error *);` – Define (but not display) a window, returning a window handle.

`int create_icon(icon_block *, error *);` – Add an icon definition to that of a window, returning an icon handle.

`void delete_wind(int w_handle, error *);` – Delete the window identified by `w_handle`.

`void delete_icon(int w_handle, int i_handle, error *);` – Delete the icon identified by `i_handle` from the window specified by `w_handle`.

`void open_wind(open_block *, error *);` – Make a window appear on the screen.

`void close_wind(int w_handle, error *);` – Remove the window identified by `w_handle` from the list of active windows.

`int poll_wimp(int mask, univ_block *, error *);` – Ask what to do next. The `mask` gives the set of disallowed return codes. The code of the next task to perform is returned as the result, and the information pertaining to it is found in the `univ_block`.

`int redraw_wind(redraw_block *, error *);` – Draw a window outline and icons. The window to be redrawn is indicated by the window handle in the `redraw_block`. The return value is false (0) if there are no rectangles to be drawn.

`int update_wind(redraw_block *, error *);` – Return the visible portion of a window for updating. If the return is false (0), there are no rectangles to update.

`int get_rectangle(redraw_block *, error *);` – Get the next rectangle in a redraw list. Returns false (0) if there are no more.

`void get_wind_state(int w_handle, open_block *, error *);` – Read the current state of the window identified by `w_handle`. State data is returned in `open_block`.

`void set_icon_state(istate_block *, error *);` – Set some/all of an icon's flags. The window and icon handles are held in `istate_block`.

`void get_icon_state(istate_block *, error *);` – Return, in `*istate_block`, the definition of the icon identified by the icon handle field of `*istate_block`.

`void get_point_info(mouse_block *, error *);` – Return data about the mouse pointer.

`void drag_box(drag_block *, error *);` – Start the wimp dragging a box.

`void force_redraw(redraw_block *, error *);` – Force a redraw of an area of the screen. Only the first five `int`s in the `redraw_block` have significance.

`void set_caret_pos(caret_block *, error *);` – Set the position and size of the text caret.

`void get_caret_pos(caret_block *, error *);` – Get the position and size of the text caret.

`void create_menu(menu_block *, int, int, error *);` – Initialise a 'pop up' menu structure.

`void decode_menu(menu_block *, univ_block *, textbuf *, error *);` – Decode a menu selection.

`void which_icon(which_block *, icon_list *, error *);` – Look for icons with particular flag settings.

`void set_extent(redraw_block *, error *);` – Alter the extent of a window's work area. Only the window handle and first set of four coordinates are used from `redraw_block`.

`void set_point_shape(pshape_block *, error *);` – Set on-screen pointer's shape.

`void open_template(char *name, error *);` – Open a file called `name` to allow `load_template()` to read a template from it.

`void close_template(error *);` – Close the currently open template file.

`void load_template(temp_block *tb, error *);` – Load a window template from the open file into a buffer pointed to by `tb->buf`. This pointer to a buffer can be cast to a `(wind_block *)` and then used to create a window using `create_wind()`.

# ASSEMBLY LANGUAGE INTERFACE

Object code modules from the Acorn C compiler can be linked with those produced by ObjAsm, provided that they observe the conventions of the *ARM Procedure Call Standard*. (ObjAsm is a variant of the ARM assembler AAsm; AAsm generates directly executable code, whereas ObjAsm generates code in a format suitable for linking, ie in Acorn Object Format.)

This chapter gives a brief description of how to handle procedure entry and exit in assembly language in order to interface to C. For details on AAsm and ObjAsm syntax and AOF files, you should consult:

*ARM Assembler* and *Archimedes Programmer's Reference Manual*, *ARM Procedure Call Standard*.

## REGISTER NAMES

The following names are used in referring to ARM registers:

| | | |
|---|---|---|
| a1 | R0 | Argument 1, also integer result |
| a2 | R1 | Argument 2 |
| a3 | R2 | Argument 3 |
| a4 | R3 | Argument 4 |
| v1 | R4 | Register variable |
| v2 | R5 | Register variable |
| v3 | R6 | Register variable |
| v4 | R7 | Register variable |
| v5 | R8 | Register variable |
| v6 | R9 | Register variable |
| fp | R10 | Frame pointer |
| ip | R11 | Used as temporary workspace |
| sp | R12 | Lower end of current stack frame |
| sl | R13 | Stack limit |
| lr | R14 | Link address on calls, or workspace |
| pc | R15 | Program counter and processor status |
| | | |
| f0 | F0 | Floating point result |
| f1 | F1 | Floating-point work register |

| | | |
|---|---|---|
| f2 | F2 | Floating-point work register |
| f3 | F3 | Floating-point work register |
| f4 | F4 | Floating-point register variable (must be preserved) |
| f5 | F5 | Floating-point register variable (must be preserved) |
| f6 | F6 | Floating-point register variable (must be preserved) |
| f7 | F7 | Floating-point register variable (must be preserved) |

In this section, 'at `[r]`' means at the location pointed to by the value in register `r`; 'at `[r,#n]`' refers to the location pointed to by `r+n`. This accords with AAsm's syntax.

## REGISTER USAGE

The following points should be noted about the contents of registers across function calls.

- Calling a function (potentially) corrupts the argument registers `a1` to `a4`, `ip`, `lr`, and `f0-f3`. The calling function should save the contents of any of these registers it may need.

- Register `lr` is used at the time of a function call to pass the return link to the called function; it is not necessarily preserved during or by the function call.

- The stack pointer `sp` is not altered across the function call itself, though it may be adjusted in the course of pushing arguments inside a function. The limit register `sl` may change at any time, but should always represent a valid limit to the downward growth of `sp`. User code will not normally alter this register.

- Registers `v1` to `v6`, and the frame pointer `fp`, are expected to be preserved across function calls. The called procedure is responsible for saving and restoring the contents of any of these registers which it may need to use.

## CONTROL ARRIVAL

At a procedure call, the convention is that the registers are used as follows:

- `a1` to `a4` contain the first four arguments.
- `sp` points to the fifth argument; any further arguments will be located in succeeding words above `[sp]`.
- `fp` points to a backtrace structure.
- `sp` and `sl` define a temporary workspace of at least 256 bytes available to the procedure.
- `lr` contains the value which should be restored into `pc` on exit from the called procedure.
- `pc` contains the entry address of the called procedure.
- `sl` contains a stack chunk handle, which is used by stack handling code to extend the stack in a non-contiguous manner.

## PASSING ARGUMENTS

All integral and pointer arguments are passed as 32-bit words. Floating point 'float' arguments are 32-bit values, 'double'-argument 64-bit values. These follow the memory representation of the IEEE single and double precision formats.

Arguments are passed *as if* by the following sequence of operations:

- Push each argument onto the stack, last argument first.
- Pop the first four words (or as many as were pushed, if fewer) of the arguments into registers `a1` to `a4`.

- Call the function, for example by the 'branch with link' instruction: `BL` *functionname*.

  In many cases it is possible to use a simplified sequence with the same effect (eg load three argument words into `a1-a3`).

  If more than four words of arguments were passed, the calling procedure should adjust the stack pointer after the call, incrementing it by four for each argument word which was pushed and not popped.

## RETURN LINK

On return from a procedure, the registers are set up as follows:

- `fp`, `sp`, `sl`, `v1` to `v6` and `f4` to `f7` have the same values that they contained at the procedure call.
- Any result other than a floating point or a multi-word structure value is placed in register `a1`.
- A floating point result should be placed in register `f0`.

Structure values returned as function results are discussed below.

## STRUCTURE RESULTS

A C function which returns a multi-word structure result is treated in a slightly different manner from other functions by the compiler. A pointer to the location which should receive the result is added to the argument list as the first argument, so that a declaration such as the following:

```
s_type afunction(int a, int b, int c)
{
      s_type d;
      /* ... */
      return d;
}
```

is in effect converted to this form:

```
void afunction(s_type *p, int a, int b, int c)
{
        s_type d;
        /* ... */
        *p = d;
        return;
}
```

Any assembler-coded functions returning structure results, or calling such functions, must conform to this model in order to interface successfully with object code from the C compiler.

## STORAGE OF VARIABLES

The code produced by the C compiler uses procedure argument values from registers where possible; otherwise they are addressed relative to `fp`, as illustrated in *Examples* below.

Local variables, by contrast, are always addressed with positive offsets relative to `sp`. In code which alters `sp`, this means that the offset for the same variable will differ from place to place. The reason for this approach is that it permits the stack overflow procedure to recover by changing `sp` and `sl` to point to a new stack segment as necessary.

## FUNCTION WORKSPACE

The values of `sp` and `sl` passed to a called function define an area of readable, writeable memory available to the called function as workspace. All words below `[sp]` and at or above `[sl,#-512]` are guaranteed to be available for reading and writing, and the minimum allowed value of `sp` is `sl-256`. Thus the minimum workspace available is 256 bytes.

The C run-time system, in particular the stack extension code, requires up to 256 bytes of additional workspace to be left free. Accordingly, all called

functions which require no more than 256 bytes of workspace should test that `sp` does not point to a location below `sl`, in other words that at least 512 bytes remain. If the value in `sp` is less than that in `sl`, the function should call the stack extension function `x$stack_overflow`. Functions which need more than 256 bytes of workspace should amend the test accordingly, and call `x$stack_overflow1`, as described below. The following examples illustrate a method of performing this test.

## EXAMPLES

The following fragments of assembler code illustrate the main points to consider in interfacing with the C compiler. If you want to examine the code produced by the compiler in more detail for particular cases, you can request an assembler listing with the compiler option `-S`.

This is a function `gggg` which expects two integer arguments and uses only one register variable, `v1`. It calls another function `ffff`.

```
      AREA    |C$$code|, CODE, READONLY
      IMPORT  |ffff|
      EXPORT  |gggg|
gggx  DCB     "gggg", 0     ;name of func., 0 terminated
      ALIGN                 ;padded to word boundary
gggy  DCD     &ff000000 + gggy - gggx
                            ;dist. to start of name
;Function entry: save necessary regs. and args. on
stack
gggg  MOV     ip, sp
      STMFD   sp!, {a1, a2, v1, fp, ip, lr, pc}
      SUB     fp, ip, #4    ;points to saved pc
;Test workspace size
      CMPS    sp, sl
      BLLT    |x$stack_overflow|
;Main activity of function
; ....
      ADD     v1, v1, 1     ;use a register variable
      BL      |ffff|        ;call another function
```

```
      CMP      v1, 99          ;rely on reg. var. after
call
; ....
;Return: place result in a1, and restore saved
registers
      MOV      a1, result
      LDMEA    fp, {v1, fp, sp, pc}^
```

If a function will need more than 256 bytes of workspace, it should replace the two-instruction workspace test shown above with the following:

```
      SUB      ip, sp, #n
      CMP      ip, sl
      BLLT     |x$stack_overflow1|
```

where n is the number of bytes needed. Note that `x$stack_overflow1` must be called if more than 256 bytes of frame are needed. `ip` must contain `sp_#needed`, as shown in the example above.

A function which expects more than four words of arguments should store its arguments in the following manner:

```
      MOV    ip, sp                  ;copy value of sp
      STMFD  sp!, {a1, a2, a3, a4};save 4 words of args.
      STMFD  sp!, {v1, v2, fp, ip, lr, pc}
                                     ;save v1-v6 needed
      SUB    fp, ip, #20             ;fp points to saved pc
      CMPS   sp, sl                  ;test workspace
      BLCC   |x$stack_overflow|
```

# ERRORS AND WARNINGS

When compiling, the compiler can produce error or warning messages of several degrees of importance. They are:

- Warnings indicating definite or possible offences against the draft ANSI standard.
- Non-serious errors which still allow code to be produced.
- Serious errors which may produce loss of code.
- Fatal errors which stop the compiler from compiling.
- System errors which signal faults in the compiler itself.

Future releases of the compiler may distinguish further errors or produce slightly different forms of wording.

The messages are listed alphabetically in each section.

## WARNINGS

Warning messages indicate legal but curious C programs, offences against the ANSI draft, or possibly unintended constructs. On detection of a warning condition, the compiler issues a warning message, if enabled, then continues compilation.

- `#define macro 'xx' defined but not used`

- `'&' unnecessary for function or array xx`
  This is a reminder that if *xx* is defined as `char xx[10]` then *xx* has type `char*`. There is a similar reminder for function names also. Example:

```
static char mesg[] = "hello\n";
int main ()
{
     char *p = &mesg; /* mesg already has type char* */
     ...
```

- `actual type 'xx' mismatches format '%x'`
  A type error in a `printf` or `scanf` format string. Example:

```
{
        int i;
        printf("%s\n", i); /* %s need char* not int */
        ...
```

- `ANSI 'xx' trigraph for 'x' found - was this intended?`
  This helps to avoid inadvertent use of ANSI trigraphs. Example:

```
printf("Type ??/!!: "); /* ??/ is trigraph for \*/
```

- `character sequence /* inside comment - error?`
  You cannot nest comments in C. Example:

```
/* comment out func() for now...
/* func() returns a random number */
int func(void)
{
        ...
        return i;
}
*/
```

- `Dangling 'else' indicates possible error`
  This hints that you may have mis-matched your `if`s and `else`s. Remember an `else` always refers to the most recent un-matched `if`. Use parentheses to avoid ambiguity. Example:

```
if (a)
   if (b)
      return 1;
   else if (c)
      return 2;
else /* this belongs to the if (a). Or does it?*/
   return 3;
```

- `Deprecated declaration of xx() - give arg types`
  A feature of the ANSI draft standard is that argument types should be given in function declarations (prototypes). 'No arguments' is indicated by `void`. Example:

```
extern int func();/* should have 'void' in the
parentheses */
```

- `extern 'main' needs to be 'int' function`
  This is a reminder that main() is expected to return an integer. Example:

```
void main()
{
      ...
```

- `floating point overflow when folding`
  This is typically caused by a division by zero in a floating point constant expression evaluated at compile time. Example:

```
#define lim 1
#define eps 0.01
static float a = eps/(lim-1); /* lim-1 yields 0 */
```

- `floating to integral conversion failed`
  A cast (possibly implicit) of a floating point constant to an integer failed at compile time. Example:

```
static int i = (int) 1.0e20; /* maxint is about 1e10 */
```

- `formal parameter 'xx' not declared - 'int' assumed`
  The declaration of a function parameter is missing. Example:

```
int func(a)
/*a should be declared here or within the parentheses*/
{
      ...
```

- `Format requires nn parameters, but mm given`
  Mismatch between a `printf` or `scanf` format string and its other arguments. Example:

```
printf("%d, %d\n",1); /* should be two ints */
```

- `function xx declared but not used`
  The (static) identifier *xx* was declared but not used within the source file.

- `illegal format conversion '%x'`
  Indicates an illegal conversion implied by a `printf` or `scanf` format string. Example:

```
printf("%w\n",10); /* no such thing as %w */
```

- `implicit return in non-void function`
  A non-void function may exit without using a return statement, but won't return a meaningful result. Example:

```
int func(int a)
{
    int b=a*10;
    .../* no return <expr> statement */
}
```

- `implicit return in non-void xx()`
  As above.

- `Incomplete format string`
  A mistake in a `printf` or `scanf` format string. Example:

```
printf("Score was %d%",score); /* 2nd % should be
                                  %% */
```

- `'int xx()' assumed - 'void' intended?`
  The definition of a function omits its return parameter type – ANSI defines the default to be `int`. You should be explicit about the type, using `void` if the function doesn't return a result. Example:

```
main()
{
      ...
```

- `inventing 'extern int xx();'`
  The declaration of a function is missing. Example:

```
      printf("Type your name: ");
      /* forgot to #include <stdio.h> */
```

- `label xx was defined but not used`
  Example:

```
errlab: exit(-1); /* this point never referenced */
```

- `no side effect in void context: 'op'`
  An expression which does not yield any side effect was evaluated; it will have no effect at run-time. Example:

```
      a+b;
```

- `non-portable - not 1 char in 'xx'`
  Assigning character constants containing other than one character to an `int` may produce non-portable results. Example:

```
static int exitCode = 'ABEX';
```

- `non-value return in a non-void function`
  The expression was omitted from a `return` statement in a function which was defined with a non-`void` return type. Example:

```
int func(int a)
{
      int b=a*10;
      ...
      return; /* no <expr>  */
}
```

- `omitting trailing '\0' for char[nn]`
  The character array being equated to a string is one character too short for the whole string, so the trailing zero is being omitted. Example:

```
static char mesg[14] = "(C)1988 Acorn\n";/* needs 15 */
```

- `Re-definition of #define macro xx`
  Redefining a macro can indicate a possible oversight.

- `Shift by nn illegal in ANSI C`
  ANSI C only allows shifts of between 0 and 31 (the size of an `int` in bits). Constant shifts outside of this range but between 32 and 255 (which are acceptable to the ARM hardware) produce this message. Shifts outside the range 0 to 255 produce the next warning. Example:

```
static int mask = 1<<32; /* would give 0 anyway */
```

- `Shift by nn illegal - here treated as nn`
  This is given for negative constant shifts or shifts greater than 255. The bottom byte of the number given is used, ie it is treated as `(unsigned char)` *nn*. NB: negative shifts are *not* treated as positive shifts in the other direction. Example:

```
printf("%d\n",1<<-2);
```

- `'short' slower than 'int' on this machine`
  For speed you are advised to use `int`s rather than `short`s where possible. This is because of the overhead of performing implicit casts from `short` to

`int` in expression evaluation. However, `shorts` are half the size of `ints`, so arrays of `shorts` can be useful. Example:

```
{
        short i,j; /* quicker to use ints */
        ...
```

- `spurious {} around scalar initialiser`
  Curly brackets are only required around structure and array initialises. Example:

```
static int i = {INIT_I}; /* don't need brackets */
```

- `static` *xx* `declared but not used`
  A static variable was declared in a file but never used in it. It is therefore redundant.

- `'struct' tag '`*xx*`' not defined`
  It is possible for a `struct` tag to be declared before the body of the structure is defined, to allow for mutually referencing structures. This warning implies that the end of the block containing the incomplete definition was encountered without the full definition being given. Example:

```
{
        struct s; /* forward reference */
        struct t {
                struct s    *sp; /* reference to struct s */
                int         i;
        }
        ...
        /* forgot to define struct s */
}
```

  Certain styles of advanced programming using 'Abstract Data Types' may generate this warning frequently in client modules which import 'opaque' types.

- `Undefined macro 'xx' in #if - treated as 0`

- `Unrecognised #pragma -x`
- `Unrecognised #pragma (no '-')`
  `#pragma` directives are of the form

  `#pragma -xd`

  where *x* is a letter an *d* is an optional digit. These messages warn against unknown letters and missing -s. Example:

  ```
  #pragma stack_check(0)  /* from some other compiler */
  ```

- `use of 'op' in condition context`
  Warns of such possible errors as = and not == in an `if` or looping statement. Example:

  ```
  if (a=b) {
      ...
  ```

- `variable xx declared but not used`
  This refers to an automatic variable which was declared at the start of a block but never used within that block. It is therefore redundant. Example:

  ```
  int func(int p)
  {
      int a;      /* this is never used */
      return p*100;
  }
  ```

- `xx treated as xxul in 32-bit implementation`
  This message warns of two's complement arithmentic's dependence on assigning negative constants to `unsigned ints`, and it explains that `ints` and `long ints` are both 32 bits.

## NON-SERIOUS ERRORS

These are errors which will allow 'working' code to be produced – they will not produce loss of code. On detection of such an error the compiler issues an error message, if enabled, then continues compilation.

- `',' (not ';') separates formal parameters`
  Incorrect punctuation between function parameters. Example:

```
extern int func(int a;int b);
```

- `'op': cast between function and object pointer`
  Casts between function and object pointers can be very dangerous!
  One possibly valid (but still very suspect) use is in casting an array of `int` into which machine code has been loaded into a function pointer. Example:

```
static int mcArray[100];
/*pointer to function returning void*/
typedef void (*pfv)(void);
    ...
   ((pfv)mcArray)();/* convert to fn type and apply */
```

- `'op': implicit cast of non-0 int to pointer`
  Zero, equal to a `NULL` pointer, is the only `int` which can be legally implicitly cast to a pointer type. Example:

```
{
     int i, *ip;
     ip = i; /* only the constant int 0 can be
                implicitly cast to a pointer type */
     ...
```

- `'op': implicit cast of pointer to non-equal pointer`
  An illegal implicit cast has been detected bewteen two different pointer types. The type casting must be made explicit to escape this error. Example:

```
{
        int   *ip;
        char  *cp;
        ip = cp; /* differing pointer types */
        ...
```

- `'op': implicit cast of pointer to 'int'`
  An illegal implicit cast has been detected between an integer and a pointer. Such casts must be made explicitly. Example:

```
{
        int i, *ip;
        i = ip; /* pointer must be cast explicitly */
        ...
```

- `'op': implicit cast of 'xx' to 'int'`
  An illegal implicit cast has been detected.

- `'register' attribute for 'xx' ignored when address taken`
  Addresses of register variables cannot be calculated, so an address being taken of a variable with a `register` storage class causes that attribute to be dropped. Example:

```
{
        register int i, *ip;
        ip = &i; /* & forces i to lose its register
                    attribute */
        ...
```

- `'union' tag 'xx' not defined`
  See above.

- `<int> op <pointer> treated as <int> op (int)<pointer>`
  Warns of an illegal implicit cast within an expression. Typically *op* is an operator which has no business being used on pointers anyway, such as `*`. Example:

```
{
        int i, *ip;
        i = i | ip; /* bitwise-or on a pointer?! */
        ...
```

- `<pointer> op <int> treated as (int)<pointer> op <int>`
  As above but with the operands the other way round.

- `ANSI C does not support 'long float'`
  This used to be a synonym for `double`, but is not allowed in ANSI C.

- `Array [0] found`
  The minimum subscript count allowed is 1. (Remember that the subscripts go from `0..n-1`.) Example:

  ```
  static int a[0];
  ```

- `assignment to 'const' object 'xx'`
  You can't assign to objects declared as `const`. Example:

  ```
  {
        const int ic = 42; /* initialisation ok */
        ic = 69; /* can't change it now */
        ...
  ```

- `comparison 'op' of pointer and int:`
  `literal 0 (for == and !=) is the only legal case`
  You cannot use the comparison operators on an integer and pointer type. As the message implies, you can only check for a pointer being (not) equal to `NULL` (`int 0`). Example:

  ```
  {
        int i,j,*ip;
        j = i>ip; /* can't compare an int and a * int */
        ...
  ```

- `Control character 0xnn found - ignored`
  An unrecognised character was found embedded in your source – this could be file corruption, so back up your sources! Note that 'control code' means any non-whitespace, non-printable character.

- `differing pointer types: 'xx'`
  An illegal implicit type cast was detected in a comparison operation between two pointers of different types. Example:

```
{
        int   *ip;
        char  *cp;
        printf("%d\n", ip==cp); /* can't compare these */
        ...
```

- `differing pointer types: '_?_:_'`
  The two sub-expressions of a `?:` conditional expression operator were pointers to different types. Example:

```
{
        int i, *ip;
        char *cp;
        printf("%p\n",i ? ip : cp); /* can't have char *
or                                      int * */
        ...
```

- `Digit 8 or 9 found in octal number`
  Octal (base 8) numbers may only have digits up to 7. Example:

```
static int i = 0178; /* probably meant 0177, ie 0xff */
```

- `formal name missing in function definition`
  This error occurs when a comma in a function definition led the compiler to suspect a further formal parameter was going to follow, but none did. Example:

```
int a(int b,) /* missing parameter */
{
     ...
```

- `function prototype formal 'xx' needs type or class - 'int' assumed`
  A formal parameter in a function prototype was not given a type or class. It needs at least one of these (`register` being the only allowed class). Example:

```
void func(a); /* I mean int a or perhaps register a */
```

- `function xx may not be initialised - assuming function pointer`
  A function is not a variable, so cannot be initialised. As an attempt to initialise *xx* has been made, *xx* is treated as of type function *. Example:

```
extern int func(void);
static int fn() = func; /* the compiler will use
                  static int (*fn)() = func; instead */
```

- `illegal string escape '\x' - treated as x`
  Unrecognised string escape (\ followed by a character) found. The \ is ignored. Example:

```
     printf("\w"); /* no such escape */
```

- `junk at end of #xx line - ignored`
  The *xx* is either `else` or `endif`. These directives should not have anything following them on the line. Example:

```
/* text after the #else should be a comment */
#else if it isn't defined
...
```

- `linkage disagreement for 'xx' - treated as 'xx'`
  There was a linkage type disagreement for declarations, eg a function was declared as `extern` then defined later in the file as `static`. Example:

```
int func(int a); /* compiler assumes extern here */
...
static func(int a) /* but told static here */
{
        ...
```

- `Missing newline before EOF - inserted`
  The last line of the source file did not have its terminating end of line character.

- `Missing type specification - 'int' assumed`

- `more than 4 chars in '...'`
  A string constant of more than four characters cannot be assigned to a 32 bit `int`. Example:

```
{
        int i = '12345'; /* more than four chars */
        ...
```

- `no chars in character constant ''`
  At leat one character should appear in a character constant. The empty constant is taken as zero. Example:

```
{
        int i = ''; /* less than one char == '\0' */
        ...
```

- `Omitted <type> before formal declarator - 'int' assumed`
  This is given in a formal parameter declaration where a type modifier is given but no base type. Example:

```
int func(*a); /* a is a pointer, but to what? */
```

- `parentheses (..) inserted around expression following 'op'`
  Parentheses were not present when needed to avoid ambiguity.

- `return <expr> illegal for void function`
  A function declared as void must not return with an expression. Example:

```
void a(void)
{
      ...
      return 0;
}
```

- `size of 'void' required - treated as 1`
  This indicates an attempt to do pointer arithmetic on a `void *`, probably indicating an error. Example:

```
{
      void  *vp;
      vp++; /* how many bytes to increment by ? */
      ...
```

- `size of a [] array required - treated as [1]`
  If an array is declared as having an empty first subscript size, the compiler cannot calculate the array's size. It therefore assumes the first subscript limit to be 1 if necessary. This is unlikely to be helpful.

```
extern int array[][10];
static int s = sizeof(array); /*can't determine this*/
```

- `size of function required - treated as size of pointer`
  The compiler cannot know the size of a function at compile time, so instead it uses the size of a `(*)()`. Example:

```
extern int func(void);
int main(void)
```

```
{
        int i = sizeof(func);
        ...
```

- `sizeof <bit field> illegal - sizeof(int) assumed`
  Bitfields do not necessarily occupy an integral number of bytes but they are always parts of an `int`, so the size of an `int` is used instead. Example:

```
struct s {
        int exp : 8;
        int mant : 23;
        int s : 1;
};

int main(void)
{
        struct s st;
        int i = sizeof(st.exp); /* can't obtain this in
                                   bytes */
...
```

- `Small (single precision) floating value converted to 0.0`
- `Small floating point value converted to 0.0`
  A floating point constant was so small that it had to be converted to 0.0. Example:

```
static float f = 1.0001e-38 - 1.0e-38; /* 1e-42 too
                                          small for
                                          float */
```

- `Spurious #elif ignored`
- `Spurious #else ignored`
- `Spurious #endif ignored`
  One of these three directives was encountered outside of any `#if` or `#ifdef` scope. Example:

```
#if defined sym
...
#endif
#else /* this one is spurious */
...
```

- `struct component xx may not be function - assuming function pointer`
  A variable such as a structure component cannot be declared to have type function, only function *. Example:

```
struct s {
     int fn();   /* compiler will use int (*fn)(); */
     char c;
};
```

- `type or class needed (except in function definition) - int assumed`
  You can't *declare* a function with neither a return type nor a storage class. One of these must be present. Example:

```
func(void); /* need, eg, int or static */
```

- `Undeclared name, inventing 'extern int xx'`
  The name *xx* was undeclared, so the ANSI default type `extern int` was used. Example:

```
int main(void) {
     int i = j; /*j has not been previously declared*/
     ...
```

- `variable xx may not be function - assuming function pointer`
  A variable cannot be declared to have type function, only function *. Example:

```
int main(void)
{
     auto void fn(void); /* treated as void
(*fn)(void);
                              */
     ...
```

## SERIOUS ERRORS

These are errors which will probably cause loss of generated code. On detection of such an error, an error message is issued, and compilation stops. The compiler will attempt to continue and produce further diagnostic messages, which are sometimes useful, but will delete the partly produced object file.

- `#error encountered "xx"`
  Source intentionally producing an error with a `#error` directive. Example:

```
#if CHAR_BIT != 8
#error This program needs eight-bit characters
#endif
```

- `#include file "xx" wouldn't open`
- `#include file <xx> wouldn't open`
  Probably caused by a spelling mistake in the file name. Example:

```
#include <stdef.h> /* missed out a 'd' */
```

- `'...' must have exactly 3 dots`
  This is caused by a mistake in a function prototype where a variable number of arguments is specified. Example:

```
extern int printf(const char *format,....); /*one . too
                                               many*/
```

- '{' of function body expected - found 'xx'
  This is produced when the first character after the formal parameter declarations of a function is not the { of the function body. Example:

```
int func(a)
int a;
      if (a) ... /* omitted the { */
```

- '{' or <identifier> expected after 'xx', but found 'yy'
  xx is typically struct or union, which must be followed either by the tag identifier or the open brace of the field list. Example:

```
struct *fred; /* Missed out the tag id */
```

- 'xx' variables may not be initialised

- 'op': cast to 'xx' illegal
- 'op': cast to non-equal 'xx' illegal
- 'op': illegal cast of 'xx' to pointer
- 'op': illegal cast to 'xx'
  These errors report various illegal casting operations. Examples:

```
struct s {
      int a,b;
};
struct t {
      float ab;
};

int main(void)
{
      int i;
      struct s s1;
      struct t s2;
/* '=': cast to 'int' illegal */
      i = s1;
/* '=': cast to non-equal 'struct' illegal */
```

```
        s1 = s2;
/* <cast>: illegal cast of 'struct' to pointer */
        i = *(int *) s1;
/* <cast>: illegal cast to 'int' */
        i = (int) s2;
        ...
```

- `'op': illegal use in pointer initialiser`
  (Static) pointer initialisers must evaluate to a pointer or a pointer constant plus or minus an integer constant. This error is often accompanied by others. Example:

```
extern int count;
static int *ip = &count*2;
```

- *xx* `may not have whitespace in it`
  Tokens such as the compound assignment operators (+= etc) may not have embedded whitespace characters in them. Example:

```
{
        int i;
        ...
        i + = 4; /* space not allowed between + and = */
        ...
```

- `<command> expected but found a '`*op*`'`
  This error occurs when a (binary) operator is found where a statement or side-effect expression would be expected. Example:

```
        if (a) /10; /* mis-placed ) perhaps? */
        ...
```

- `<expression> expected but found '`*op*`'`
  Similar to above. An operator was found where an operand might reasonably be expected. Example:

```
        func(>>10); /* missing left hand side of >> */
```

- `<identifier> expected but found 'xx' in 'enum' definition`
  An unexpected token was found in the list of identifiers within the braces of an `enum` definition. Example:

```
enum colour {red, green, blue,;}; /* spurious ; */
```

- `\<space> and \<tab> are invalid string escapes`
  Use `<space>` and `\t` respectively for these characters in strings and character constants. Example:

```
printf("\ Next?"); /* No need for \ */
```

- `{} must have 1 element to initialise scalar or auto`
  When a scalar (integer or floating type) is initialised, the expression does not have to be enclosed in braces, but if they are present, only one expression may be put between them. Example:

```
static int i = {1,2}; /* which one to use? */
```

- `Array size nn illegal - 1 assumed`
  Arrays have a maximum dimension of `0xffffff`. Example:

```
static char dict[0x1000000]; /* Too big */
```

- `attempt to apply a non-function`
  The function call operator `()` was used after an expression which did not yield a pointer to function type. Example:

```
{
    int i;
    i();
    ...
```

- `auto array 'xx' may not be initialised`

Automatic arrays (defined in a function) may not have initialisers. Only `static` arrays may. Example:

```
int main()
{
      int a[] = {1,2,4,8,16,32,64,128};
      ...
```

- `Bit fields do not have addresses`
  Bitfields do not necessarily lie on addressable byte boundaries, so the `&` operator cannot be used with them. Example:

```
struct s {
      int h1,h2 : 13;
};

int main(void)
{
      struct s s1;
      short *sp = &s1.h2; /*can't take & of bit field*/
      ...
```

- `Bit size` *nn* `illegal - 1 assumed`
  Bitfields have a maximum permitted width of 32 bits as they must fit in a single integer. Example:

```
struct s {
      int f1 : 40; /* This one is too big */
      int f2 : 8;
};
```

- `'break' not in loop or switch - ignored`
  A break statement was found which was not inside a `for`, `while` or `do` loop or `switch`. This might be caused by an extra `}`, closing the statement prematurely. Example:

```
int main(int argc)
{
      if (argc == 1)
            break;
      ...
```

- `'case' not in switch - ignored`
  A `case` label was found which was not inside a `switch` statement. This might be caused by an extra `}`, closing the `switch` statement prematurely. Example:

```
void fn(void)
{
      case '*': return;
      ...
```

- `'continue' not in loop - ignored`
  A `continue` statement was found which was not inside a `for`, `while` or `do` loop. This might be caused by an extra `}`, closing the `switch` statement prematurely. Example:

```
      while (cc) {
      if (dd)                    /* intended a { here */
                  error();
            }                    /*this closes the while */
            if (ee)
                  continue;
      }
```

- `'default' not in switch - ignored`
  A `default` label was found which was not inside a `switch` statement. This might be caused by an extra `}`, closing the `switch` statement prematurely. Example:

```
      switch (n) {
            case 0:
```

```
            return fn(n);
        case 1: if (cc)
            return -1;
                else
             break;
        } /* spurious } closes the switch */
        default:
            error();
}
```

- `Digit required after exponent marker`
  A syntax error in a floating point constant was found. Example:

```
a = b*1.1e; /* need [+/-]digits here */
```

- `duplicated case constant: nn`
  The case label whose value is nn was found more than once in a `switch` statement. Note that nn is printed as a decimal integer regardless of the form the expression took in the source. Example:

```
switch (n) {
        case ' ':
        ...
        case ' ':
        ...
}
```

- `duplicate 'default' case ignored`
  Two cases in a single `switch` statement were labelled `default`. Example:

```
switch (n) {
        default:
        ...
        default:
        ...
}
```

- `duplicate definition of 'struct' tag 'xx'`
  There are duplicate definitions of the type `struct xx {...} ;`.
  Example:

```
struct s { int i,j;};
struct s {float a,b;};
```

- `duplicate definition of 'union' tag 'xx'`
  There are duplicate definitions of the type `union xx {...} ;`. Example:

```
union u {int i; char c[4];};
union u {double d; char c[8];};
```

- `duplicate declaration of 'xx'`
- `duplicate definition of 'xx'`
- `duplicate definition of label xx -ignored`
  These all refer to various types of duplicated definition. Examples:

```
static int i;
void fn(void)
{
     lab:
     ...
     lab:        /* redefinition of lab */
}
char i;          /* redefinition of i */
int fn()         /* redefinition of fn() */
{
     ...
}
```

- `duplicate type specification of formal parameter 'xx'`
  A formal function parameter had its type declared twice, once in the argument list and once after it. Example:

```
void fn(int i)
int i;          /* this one is redundant */
{
     ...
```

- `EOF in comment`
- `EOF in string`
- `EOF in string escape`
- `EOF not newline after #if...`

  These all refer to unexpected occurrences of the end of the source file.

- `Expected <identifier> after 'xx' but found 'xx'`

- `expected 'xx' or 'x' - inserted 'x' before 'yy'`
- `expected 'xx' - inserted before 'yy'`

  This typically occurs when a terminating semi-colon has been omitted before a }. (Common amongst Pascal programmers) Another case is the omission of a closing bracket of a parenthesised expression. Examples:

```
int fn(int a, int b, int c)
{
     int d = a*(b+c;   /* missing ) */
     return d          /* missing ; */
}
```

- `Expecting <declarator> or <type>, but found 'xx'`

  xx is typically a punctuation character found where a variable or function declaration or definition would be expected (at the top level). Example:

```
static int i = MAX;+1; /* spurious ; ends expression */
```

- `'goto' not followed by label - ignored`

  Self explanatory.

- `Grossly over-long floating point number`
  Only a certain number of decimal digits are needed to specify a floating point number to the accuracy that it can be stored to. This number of digits was exceeded by an unreasonable amount.

- `Grossly over-long hexadecimal constant`
  A hexadecimal constant has an excessive number of leading zeros, not affecting its value.

- `Grossly over-long number`
  A constant has an excessive number of leading zeros, not affecting its value.

- `Grossly over-long string`

- `Hex digit needed after 0x or 0X`
  Hexdecimal constants must have at least one digit from the set `0..9`, `a..f`, `A..F` following the `0x`. Example:

```
int i = 0xg; /* illegal hex char */
```

- `Identifier (xx) found in <abstract declarator>`
  The `sizeof()` function and cast expressions require abstract declarators, ie types without an identifier name. This error is given when an identifier is found in such a situation. Examples:

```
i = (int j) ip; /* trying to cast to integer */
l = sizeof(char str[10]); /* probably just mean
                             sizeof(str) */
```

- `illegal bit field type 'xx' - 'int' assumed`
  `Int` (signed or unsigned) is the only valid bitfield type in ANSI-conforming implementations. Example:

```
struct s { char a : 4; char b : 4;};
```

- `illegal in #if <expression>: xx`
- `illegal in case expression (ignored): xx`

- `illegal in constant expression: xx`
- `illegal in floating type initialiser: xx`

  All of these errors occur when a constant is needed at compile time but a variable expression was found.

- `Illegal in l-value: 'enum' constant 'xx'`

  An incorrect attempt was made to assign to an `enum` constant This could be caused by mis-spelling an `enum` or variable identifier. Example:

```
enum col {red, green, blue};
int fn()
{
      int read;
      red = 10;
      ...
```

- `Illegal in the context of an l-value: 'xx'`
- `Illegal in lvalue: function or array 'xx'`

  An incorrect attempt was made to assign to *xx*, where the object in question is not assignable (an l-value). You can't, for example, assign to an array name or a function name. Examples:

```
{
      int a,b,c;
      a ? b : c = 10;   /* ?: can't yield l-values. */
      if (a)            /* use this instead */
            b = 10;
      else
            c = 10;
      ...
```

or, in the same context,

```
      *(a ? &b: &c) = 10;
```

- `illegal in static integral type initialiser: xx`
  A constant was needed at compile time but a suitable expression wasn't found.

- `Illegal types for operands : 'op'`
  An operation was attempted using operands which are unsuitable for the operator in question. Examples:

```
{
      struct {int a,b;} s;
      int i;
      i = *s;      /* can't indirect through a struct */
      s = s+s;     /* can't add structs */
      ...
```

- `Junk after #if <expression>`
- `Junk after #include "xx"`
- `Junk after #include <xx>`
  None of these directives should have any other non-whitespace characters following the expression/filename. Example:

```
#include <stdio.h> this isn't allowed
```

- `label 'xx' has not been set`
  An attempt has been made to use a label that has not been declared in the current scope, after having been referenced in a `goto` statement. Example:

```
int main(void)
{
      goto end;
}
```

- `Misplaced '{' at top level - ignoring block`
  { } blocks can only occur within fuction definitions. Example:

```
/* need a function name here */
{
        int i;
        ...
```

- `Misplaced 'else' ignored`
  An `else` with no matching `if` was found. Example:

```
        if (cc)             /* should have used { } */
                i = 1;
                j =2;
        else
                k = 3;
        ...
```

- `Missing #endif at EOF`
  A `#if` or `#ifdef` was still active at end of the source file. These directives must always be matched with a `#endif`.

- `Missing '"' in pre-processor command line`
  A line such as `#include "`*name* has the second " missing.

- `Missing ')' after` *xx*`(... on line` *nn*
  The closing bracket (or comma separating the arguments) of a macro call was omitted. Example:

```
#define rdch(p) {ch=*p++;}
...
{
        rdch(p      /* missing ) */
        ...
```

- `Missing ',' or ')' after #define` *xx*`(...`
  One of the above characters was omitted after an identifier in the macro parameter list. Example:

```
#define rdch(p {ch = *p++;}
```

- `Missing '<' or '"' after #include`
  A `#include` filename should be within either double quotes or angled brackets.

- `Missing identifier after #define`
- `Missing identifier after #ifdef`
- `Missing identifier after #undef`
  Each of these directives should be followed by a valid C identifier. Example:

```
#define @ at
```

- `Missing parameter name in #define xx(...`
  No identifier was found after a `','` in a macro parameter list. Example:

```
#define rdch(p,) {ch=*p++;}
```

- `Missing hex digit(s) after \x`
  The string escape `\x` is intended to be used to insert ASCII coded characters in a string, but was incorrectly used here. It should be followed by between one and three hexadecimal digits. (On the present system, if there are three digits following the `\x`, the first one is ignored.) Example:

```
printf("\xxx/"); /* probably meant "\\xxx/" */
```

- `No ')' after #if defined (...`
  The `defined` operator expects an identifier, optionally enclosed within brackets. Example:

```
#if defined(debug
```

- `No identifier after #if defined`
  See above.

- `non static address 'xx' in pointer initialiser`
  An attempt was made to take the address of an automatic variable in an expression used to initialise a `static` pointer. Such addresses are not known at compile-time. Example:

```
{
        int i;
        static int *ip = &i; /*&i not known to compiler*/
        ...
```

- `Non-formal 'xx' in parameter-type-specifier`
  A parameter name used to declare the parameter types did not actually occur in the parameter list of the function. Example:

```
void fn(a)
int a,b;
{
        ...
```

- `Number nn too large for 32-bit implementation`
  An integer constant was found which was too large to fit in a 32 bit `int`. Example:

```
static int mask = 0x800000000; /*0x80000000 intended?*/
```

- `Overlarge floating point value found`
- `Overlarge (single precision) floating point value found`
  A floating point constant has been found which is so large that it will not fit in a floating point variable. Examples:

```
float f = 1e40; /* largest is approx 1e38 for float */
double d = 1e310; /* and 1e308 for double */
```

- `quote (" or ') inserted before newline`
  Strings and character constants are not allowed to contain unescaped newline characters. Use `\<nl>` to allow strings to span lines. Example:

```
        printf("Total =
```

- `re-using 'struct' tag 'xx' as 'union' tag`
  There are conflicting definitions of the type `struct xx {...} ;` and `union xx {...} ;`. Structure and union tags currently share the same name-space in C, although this not actually necessary. Example:

```
struct s {int a,b;};
...
union s {int a; double d;};
```

- `re-using 'union' tag 'xx' as 'struct' tag`
  As above.

- `size of struct 'xx' needed but not yet defined`
  An operation requires knowledge of the size of the struct, but this was not defined. This error is likely to accompany others. Example:

```
{
        struct s;           /* forward declaration */
        struct s *sp;       /* pointer to s */
        sp++;               /* need size for inc operation*/
        ...
```

- `size of union 'xx' needed but not yet defined`
  See above.

- `storage class 'xx' incompatible with 'xx' - ignored`
  An attempt was made to declare a variable with conflicting storage classes. Example:

```
{
        static auto int i; /* contradiction in terms */
        ...
```

- `storage class 'xx' not permitted in context xx - ignored`
  An attempt was made to declare a variable whose storage class conflicted with its position in the program. Examples:

```
register int i;  /* can't have top-level regs */
void fn(a)
static int a;    /* or static parameters */
{
     ...
```

- `string initialiser longer than char[nn]`
  An attempt was made to initialise a character array with a string longer than the array. Example:

```
static char str[10] = "12345678901234";
```

- `struct 'xx' must be defined for (static) variable declaration`
  Before you can declare a static structure variable, that structure type must have been defined. This is so the compiler knows how much storage to reserve for it. Examples:

```
static struct s s1;    /* s not defined */
struct t;
static struct t t1;    /* t not defined */
```

- `struct/union 'xx' has no xx field`
  The field name used with a . or -> operator is not a valid one for the union or structure type 'xx' being referenced. Example:

```
struct s {int a,b;};
...
{
     struct s s1;
     s1.c = 3;   /* no c field */
     ...
```

- `struct/union 'xx' not yet defined - cannot be selected from`
  The structure or union type used as the left operand of a . or -> operator has not yet been defined so the field names are not known. Example:

```
{
        struct s s1;
        s1.a = 12;  /* don't know field names yet */
        ...
```

- `Too few arguments to macro xx(... on line nn`
- `Too many arguments to macro xx(... on line nn`
  The number of arguments used in the invocation of a macro must match exactly the number used when it was defined. Example:

```
#define rdch(ch,p) while((ch = *p++)==' ');
...
        rdch(ptr);  /* need ptr and ch */
        ...
```

- `too many initialisers in {} for aggregate`
  The list of constants in a static array or structure initialiser exceeded the number of elements/fields for the type involved. Example:

```
static int powers[8] = {0,1,2,4,8,16,32,64,128};
```

- `Too many arguments in function call - ignored`

- `type 'xx' inconsistent with 'xx'`

- `type disagreement for 'xx'`
  Conflicting types were encountered in function declaration (prototype) and its definition. Example:

```
void fn(int);
...
int fn(int a)
{
      ...
```

- `typedef name 'xx' used in expression context`
  A `typedef` name was used as a variable name. Example:

```
typedef char flag;
...
{
      int i = flag;
```

- `undefined struct/union 'xx' cannot be member`
  A `struct/union` not already defined cannot be a member of another `struct/union`. In particular this means that a `struct/union` cannot be a member of itself – circular types are not allowed. Example:

```
struct s1 {
      struct s2 type; /* s2 not defined yet */
      int count;
};
```

- `Unknown directive : #xx`
  The identifier following a # did not correspond to any of the recognised pre-processor directives. Example:

```
#asm  /* not an ANSI directive */
```

- `wrong number of parameters to 'xx'`
  The function *xx* was called with the wrong number of parameters, as defined by its protype. Example:

```
size_t strlen(const char *s);
...
```

```
{
        int i = strlen(str,j); /* only str needed */
```

- `Uninitialised static [] arrays illegal`
  Static [] arrays must be initialised to allow the compiler to determine their size. Example:

```
static char str[];      /* needs {} initialiser */
```

- `union variable 'xx' must be defined for (static) variable declaration`
  Before you can declare a static union variable, that union type must have been defined. Example:

```
static union u ul; /* compiler can't ascertain size */
```

- `'while' expected after 'do' - found 'xx'`
  The syntax of the do statement is do *statement* while (expression). Example:

```
do      /* should put these statements in {} */
        l = inputLine();
        err = processLine(l);/*finds err, not while
                                */
while (!err);
```

## FATAL ERRORS

These are causes for the compiler to give up compilation. Error messages are issued and the compiler stops.

- `Couldn't create object file 'file'`
  The compiler was unable to open or write to the specified output code file, perhaps because it was locked or the o directory does not exist.

- `macro args too long`

Grossly over-long macro arguments.

- `macro expansion buffer overflow`
  Grossly over-complicated macros were used.

- `No store left`
  `out of store (in cc_alloc)`
  The compiler has run out of memory – either shorten your source programs or free some RAM using the `*UNPLUG` command. To do this, first check which modules are present in your machine by typing `*MODULES`. If there is a module that you do not currently need, you can release its space by typing

  `*UNPLUG` *`modulename`*

  It can later be restored using the `*RMREINIT` command. For further details, refer to the section entitled *Modules* in the *Archimedes Programmer's Reference Manual, Vol 2.*

- `Too many errors`
  More than 100 serious errors were detected.

- `Too many file names`
  An attempt was made to compile too many files at once. 25 is the maximum that will be accepted.

## SYSTEM ERRORS

There are some additional error messages that can be generated by the compiler if it detects errors in the compiler itself. It is very unusual to encounter this type of error. If you do, note the circumstances under which the error was caused and contact your Acorn dealer

# PCC COMPATIBILITY MODE

This chapter discusses the differences apparent when the compiler is used in 'PCC' mode. When given the `-pcc` command line flag, the C compiler will accept (Berkeley) Unix-compatible C, as defined by the implementation of the Portable C Compiler and subject to the restrictions which are noted below.

In essence, PCC-style C is 'Kernighan and Ritchie' C, as defined in the book *The C Programming Language*, by B. Kernighan and D. Ritchie (K&R), with a small number of extensions and clarifications of language features that the book leaves undefined.

## LANGUAGE AND PRE-PROCESSOR COMPATIBILITY

In `-pcc` mode, the Acorn C compiler accepts K&R C , but it does not accept many of the old-style compatibility features, the use of which has been deprecated and warned against for many years. Differences are listed briefly below:

- Compound assignment operators where the = sign comes first are accepted (with a warning) by some PCCs. An example is =+ instead of +=. Acorn C does *not* allow this ordering of the characters in the token.

- The = sign before a `static` initialiser was not required in some very old (pre-K&R C) compilers. Acorn C does not support this syntax.

- The following, very peculiar, usage is found in some Unix tools pre-dating Unix Version 7:

```
struct {int a, b;};
double d;

d.a = 0;
d.b = 0x....;
```

This is accepted by Unix PCCs and may cause problems when porting old (and badly written) code.

- enums are less strongly types than is usual under PCCs. enum is a non-K&R extension to C which has been standardised by ANSI somewhat differently from the usual PCC implementation. The compiler warns of this when an enum is encountered.

- chars are signed by default in -pcc mode.

- In -pcc mode, the compiler permits the use of the ANSI '...' notation which signifies that a variable number of formal arguments follow.

- With the exception of enums, the compiler's type checking is generally stricter than PCC's – much more akin to lint's, in fact. In writing the Acorn C compiler, we have attempted to strike a balance between generating too many warnings when compiling known, working code, and warning of poor or non-portable programming practices. Many PCCs silently compile code which has no chance of executing in just a slightly different environment. We have tried to be helpful to those who must port C among machines in which the following varies:

  the order of bytes within a word (e.g. little-endian ARM, VAX, Intel versus big-endian Motorola, IBM370),

  the default size of int (four bytes versus two bytes in many PC implementations),

  the default size of pointers (not always the same as int),

  whether values of type char default to signed or unsigned char,

  the default handling of undefined and implementation-defined aspects of the C language.

  If the verbosity of cc -pcc is found undesirable then all warnings can be turned off by using the -w command-line flag.

- The compiler's preprocessor is believed to be equivalent to Unix's cpp, except for the points listed below. Unfortunately, cpp is only defined by its

implementation, and although equivalence has been tested over a large body of Unix source code, completely identical behaviour cannot be guaranteed. Some of the points listed below only apply when the `-E` option is used with the `cc` command.

There is a different treatment of whitespace sequences (benign).

\<nl> is processed by `cc -E`, but passed by cpp (making lines longer than expected; `cc -E` only).

Cpp breaks long lines at a token boundary; `cc -E` doesn't (this may break line-size constraints when the source is later consumed by another program `cc -E` only).

The production of `#line "file"` directives is different (`cc -E` only).

The handling of unrecognised `#` directives is different (this is mostly benign).

Literal control characters cause `cc` to issue a warning (but they are not ignored as they are in ANSI mode).

The 'whitespace control characters' form-feed, carriage-return, and vertical-tab are converted to newline.

## STANDARD HEADERS AND LIBRARY

Use of the compiler in `-pcc` mode precludes neither the use of the standard ANSI headers built in to the compiler nor the use of the run-time library supplied with the C compiler.

Of course, the ANSI library does not contain the whole of the Unix C library, but it does contain almost all the commonly used functions. Beware, however, that some functions have different names – for example the ANSI function `strrchr()` is called `rindex()` under BSD Unix.

Also some functions have a slightly different definition. For example, the ANSI `sprintf()` returns the number of characters 'printed', whereas the BSD `sprintf()` returns a pointer to the start of the character buffer. (This does not matter if the function is used in a `void` context, as is often the case.) Fortunately, the important functions such as `printf()`, `fprintf()`, `malloc()`, `strcpy()`, etc., are all identical.

Another hazard is that the 'standard' definitions are sometimes in different 'standard' places. For example, the ANSI string-handling functions are defined in `<string.h>`, whereas the BSD string-handling functions are defined in `<strings.h>`.

Unless the user directs otherwise using `-j`, the C compiler will attempt to satisfy references to, say, `<stdio.h>` from its in-store filing system. Of course, this header is an ANSI header and is invalid in `-pcc` mode. However, the compiler is able to switch itself temporarily into ANSI mode when reading in-store headers, switching back to `-pcc` mode for everything else. This behaviour cannot be exploited to use real ANSI header files: the compiler can only do it using the files it knows about (and for which the switch to ANSI mode is known to be harmless).

Time functions are very system-dependent, but this is well-known within the C community and doesn't usually constitute much of a problem in software written with portability in mind.

Notwithstanding these problems, many simple programs will compile immediately using `cc -pcc`.

# CALLING OTHER PROGRAMS FROM C

## INTRODUCTION

The C library procedure `system()` provides the means whereby a program can pass a command to the host system's command line interpreter. The semantics of this are undefined by the draft ANSI standard.

Arthur distinguishes two kinds of commands, which we can loosely term built-in commands and applications. These have different effects. The former always return to their callers; the latter return to the previously set-up 'exit handler'. Because of these differences, `system()` exhibits three kinds of behaviour. This is explained below.

Applications in Arthur are loaded at a fixed address specified by the application image. Normally, this is the base of application workspace, 0x8000. While executing, applications are free to use store between the base and end of application workspace. The end is the value returned by SWI `OS_GetEnv`. They terminate with a call of SWI `OS_Exit`, which transfers control to the current exit handler.

When a C program makes the call `system("`*command*`")`, the result depends on the type of *command*:

- A built-in command will execute and return to the C program. If there was no error, `system()` returns 0, otherwise it returns something non-0 (which may vary from one release to another).

- An application will load, possibly overwriting the C program, execute, then return through the exit handler. This is usually set to the Arthur command mode, but some applications such as Twin and BASIC's 'shell' library also set up exit handlers. The C program effectively loses control the moment it does the `system()`.

To provide finer control, two variants of `system()` are provided. If the argument to `system()` begins with `CALL:` and the callee is a built-in command or an application satisfying certain conditions (described below)

then return from `system()` is guaranteed. Similarly, if the command string begins `CHAIN:` then there will be no return, even if the chained command is built-in.

In order for an application to invoke other applications and have control return, it is necessary to:

- arrange that the invoking application occupies an address range clear of that required by the invoked application,
- set the end of application workspace below the base of the invoking application, and
- set an exit handler so that the invoking application regains control when the invoked application exits.

Note: It is not possible to load Arthur modules from an Arthur application unless the relocatable module area (RMA) has been configured to be sufficiently large. Whereas loading modules from the Arthur command mode succeeds because it can extend the RMA, this cannot be done if there is an active application.

When the library function `system()` is given an argument string starting with the characters `CALL:`, it automatically performs the second and third of these tasks, before handing the remainder of the argument string to `OS_CLI` to execute. So the C program only has to worry about the first point, avoiding a clash of memory requirements. This can be solved in two ways:

- The invoking application can be linked to load at a fixed high address. This is rather inflexible. For example, selecting an address which will allow the application to run on a range of machines with differing memory sizes means wasting memory in the larger machines.
- The (relocatable) invoking application can re-load itself at the top of available memory. This technique has the advantage of being more general than the first.

The invoking application can be linked to be (load-time) relocatable, using the linker's `-relocatable` flag. Such an application will be loaded at 0x8000 if run directly, but either the application itself can observe that it is running at too low an address, and reload itself a suitable distance from the top of memory, or one can run a small bootstrap application which loads the main application a suitable distance from the top of memory.

(One might think that in the first case, the image could be moved up from its current position to a suitable one, then re-entered. Unfortunately, by the time the image is entered the relocation table has been overwritten, so re-entering fails).

Which of these methods is preferable depends principally on the size of the application: the larger it is, the more attractive the bootstrap method. The code required is much the same in both cases, and is illustrated by a small CLI example (which uses the self-reload method) below.

Note that, in this example, when the application is reloaded, its arguments start at `argv[2]`, not `argv[1]`: `argv[0]` is `"go"` and `argv[1]` is `"<address>;"`.

Although in this example the required heap size is a constant, the method is obviously adaptable to more complicated values, for example it could use a fixed proportion of available memory, or an amount determined by (some property of) the arguments.

## AN EXAMPLE PROGRAM USING `system()`

The following example can be found on the distribution disc with the filename `HowToCall`. It is a command line interpreter that reloads itself a fixed distance from the end of store, then prompts for commands which are passed to Arthur for execution via `system()`. Any command – including the invocation of another application (eg the CLI itself)– can be executed, provided that there is enough memory.

```
#include <kernel.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define OS_GetEnv 0x10
#define MaxHeapSize (32*1024)
#define LoadFileUsingPathVariable 0xe

static void CheckError(int status)
{
  if (status==_kernel_ERROR) {
    _kernel_oserror *err = _kernel_last_oserror();
    fprintf(stderr, "can't reload: %s\n",
                                  &(err->errmess));
    exit(1);
  } else if (status!=1) {
    fprintf(stderr, "can't reload\n");
    exit(1);
  }
}

static void SelfReload(char *name)
{
/* This procedure reloads the current application a
distance below the top of memory which allows its heap
to be of size MaxHeapSize, then enters it with the same
arguments as it was given this time (and does not
return).
Except: if the application is already at the right
address, the procedure does nothing and returns.
        if the reload fails, in which case a message
saying so is produced and execution terminates.*/
  _kernel_swi_regs regs;
  int *heapptr = malloc(4);
  _kernel_swi(OS_GetEnv, &regs, &regs);
  /* (No error possible)
```

```
    * regs.r[0] is our argument string
    * regs.r[1] is the top of available memory
    */
  if ((int) heapptr < regs.r[1]-MaxHeapSize) {
    /* we are running too low in store:
     * reload ourselves higher
     */
    _kernel_osfile_block osf;
    /* reload so that the end of the word allocated
     * for heapptr will be MaxHeapSize below the top
     * of allowed memory
     */
    int mybase = 0x8000;
    int newbase = regs.r[1]-MaxHeapSize-
                          ((int) heapptr - mybase);
    char s[256];
    char *args = (char *) regs.r[0];
    while (*args!=' ') args++;
    /* skip over the application name in
     * our argument string
     */
    osf.load = newbase;
    osf.exec = 0;  /* load at our provided address */
    osf.start = (int) "run$path";
    CheckError(
 _kernel_osfile(LoadFileUsingPathVariable, name, &osf)
    );
    sprintf(s, "CHAIN:go %x; %s", newbase, args);
    system(s);
  }
  free(heapptr);
}

static int cli(void)
{
  for (;;) {
```

```
    char b[256];
    int ch, i;
    _kernel_oserror *err;

    err = NULL;
    /* prompt for input... */
    printf("cli> ");
    /* skip leading spaces */
    while ((ch = getc(stdin)) == ' ');
    i = 0;
    while (ch != '\n') {
      /* EOF means kill cli() */
      if (ch == EOF) {printf("\n");  return(0);}
      b[i++] = ch;
      ch = getc(stdin);
    }
    b[i] = 0;
    if (i != 0) {
      char c[256];
      int res;
      strcpy(c, "CALL:");
      strcat(c, b);
      res = system(c);
      if (res == 0) err = _kernel_last_oserror();
    }
    if (err!=NULL) printf("error %i: %s\n",
                          err->errnum,
                          &(err->errmess));
  }
  return(0);
}

int main(int argc, char *argv[])
{
  SelfReload(argv[0]);
  return cli();
}
```

# USING THE LINKER

The Linker is an essential program for anyone developing programs in a high-level compiled language on the Archimedes personal workstation. Its purpose is to combine the contents of one or more object files (the output of a compiler or Assembler) with one or more library files, producing a final executable program.

## SYNTAX

The format of the Link command is:

`Link -output` *file* `[`*options*`]` *files*

The *files* argument is a list of input files; this is described below. `-output` is the only compulsory keyword.

Below is a list of the command line options that the Linker can take. Most of these will only be used occasionally. In the descriptions below, the important, frequently-used options are given first, followed by the less common ones. Capitals are used to denote the alternative shortened form of the keyword.

| | |
|---|---|
| `-Output` | Name of the linked output file |
| `-VIA` | Use a file to obtain (further) input file names |
| `-Case` | Make matching of symbols case insensitive |
| `-Base` | Set base address for output file |
| `-Verbose` | Print messages indicating progress of the link operation |
| `-Relocatable` | Generate relocatable output file |
| `-Dbug` | Generate an AOF image for use with the Dbug program |

### Notes

- The keyword `-base` is followed by a numeric argument. You can use the prefix `&` to specify hexadecimal, and the suffixes `k` for $2^{10}$ and `m` for $2^{20}$.

- The default base address for the output file is &8000 (32K). If `-dbug` is specified, the default base address is &50000 (ie 320K).

- The item *files* above is a list of one or more filenames, separated by spaces. This part of the command must be given. Each of the files in the list must be in Acorn Object Format (compiled files) or Acorn Library Format (libraries). They may contain references to external objects (procedures and variables) which the Linker will attempt to resolve by matching them against definitions found in other files.

- You can use wildcards in the filename list. Names using wildcards will be expanded into the list of files matching the specification. For example, the name `o.bas*` might yield `o.basmain`, `o.basexpr`, `o.bascmd`.

- Usually, at least one library file will be specified in the list. A library is just a collection of AOF files stored in a single Acorn Library Format file. You can call the procedures in the library for one language from programs written in another, as long as both languages conform to the ARM Procedure Calling Standard and both run-time libraries use the common run-time kernel. For example, an assembler program could use the C `printf` function, as long as the C run-time system had been initialised, through the common run-time kernel.

- Libraries differ from object files in the way the Linker uses them. Object files' symbols are scanned only once when the Linker attempts to resolve external references. Libraries are scanned as many times as necessary. If a required symbol is found in one of the libraries' component files, the whole component is incorporated into the output file.

- Two common errors given during a link are caused by unresolved and multiple references. In the first case, a symbol has been referenced from a file (whose name is given in the error), but there is no corresponding definition for the symbol. This is usually caused by the omission of a required object or library file from the list, or the misspelling of a symbol in the original source program.

- The second error is caused by a clash of names. For example, a procedure might have been defined with the same name as a library procedure, or as a procedure in another object file.

- The `-output` keyword is obligatory. It is followed by the name of the file to which the final linked program should be written. If you just want to use Link to check object files for unresolved references, you can specify the device `null:` as the output file; the final object code will be discarded. The output is usually in Arthur Image Format, which can be executed directly. An alternative format allows low-level debugging with Dbug.

### Simple examples

Before we move on to describe the rest of the Link command's options, we give some examples using the syntax described so far.

```
Link -OUTPUT test.sieve aof.sieve, paslib
Link -o %.mybasic aof.bas* lib.f77
Link -o null: aof.comp*
```

## VIA KEYWORD

Sometimes you may want to link a large number of input files which would be tedious to type on a command line, and whose names can't conveniently be matched by a wildcard specification. To solve this problem, you can store a list of input filenames in another file and use the `-via` keyword to give the Linker access to them. For example, suppose you created the file `basfiles` with the contents:

```
aof.main
aof.expr
aof.cmd
aof.stmnt
aof.lex
aof.filing
```

```
aof.tokens
```

If you then used the command

```
*link -o basic -via basfiles lib
```

then the files listed in `basfiles` would be linked, together with the AOF file `lib`.

## CASE KEYWORD

If you specify `-case` in the command line, then the Linker will not treat the case of letters as significant in identifiers. By default, the identifiers main and Main refer to different objects, as they are spelt differently. However, with
`-case` set, they are the same identifier.

One reason for using this flag is if you are linking a C object file with one from a language such as ISO-Pascal or Fortran-77, both of which are case insensitive. These languages plant symbols in object files in upper case, regardless of how they are spelt in the source file.

## BASE KEYWORD

By default, the base address of the output file of the Linker is &8000. This corresponds to the start of application workspace on the Archimedes computer. Alternatively, if the `-dbug` option is given, the base address is set to &50000. This is so that the debugger program Dbug can load at &5000 as a normal application, and load the file to be debugged above itself. (There are other changes when `-dbug` is given, as described below.)

Using the `-base` keyword, you can set the base address of the output file to any desired value. For example, you may want a program to have a high load address (as with the `-dbug` option set), but still be directly executable (which a dbug file in AOF format isn't).

The keyword is followed by a number given the base address desired for the output file, eg `-base &80000`, `-base 256k` etc. When this is done, all relocatable objects in the input files are relocated using that base instead of the default.

## VERBOSE KEYWORD

If you specify `-verbose` on the command line, the Linker gives a report of its progress. A message is printed as each file is opened and as each module is being relocated. For example:

```
link: opening p.basic
link: opening o.bas1
link: opening o.bas2
link: relocating module o.bas1
link: relocating module o.bas2
link: relocating module ansilib (fpprintf)
...
```

## RELOCATABLE KEYWORD

Usually, when an image file is produced, it will execute correctly only at the base address given (or the default). This is because the program will contain references to absolute addresses within itself. However, if you specify the -relocatable option, the final program will be relocatable. That is, it can be loaded and executed at any address.

This feat is achieved by adding a relocation table and a small program to perform the relocation to the image. The relocation table is a list of offsets from the start of the program to words which need relocating. These words are adjusted by the difference between the base address of the program and the address where it was loaded. Once the relocation has been performed, the program proper starts executing.

Note that although this ability can be used to make a program statically relocatable, it does not confirm true position-independence on the program.

That is, the program could not be moved in memory once it has started and still be expected to work.

## DBUG KEYWORD

If a program is linked using the `-dbug` keyword, an executable image is not formed. Instead, an AOF file is created which contains all of the symbols found in the original source files. The code segment of the file can be executed under the control of a Dbug program, and the contents of the code and data segments may be examined (and altered in the case of the data segment).

## PREDEFINED LINKER SYMBOLS

There are several symbols which the Linker knows about independently of any of its input files. These start with the string `Image$$` and, along with all other external names containing `$$`, are reserved by Acorn.

The symbols are

| | |
|---|---|
| `Image$$RO$$Base` | Address of the start of the read-only (code) area |
| `Image$$RO$$Limit` | Address of the byte beyond the end of code area |
| `Image$$RW$$Base` | Address of the start of the read/write (data) area |
| `Image$$RW$$Limit` | Address of the byte beyond the end of the data area |

Although it will often be the case, Acorn do not guarantee that the end of the read-only area corresponds to the start of the read/write area. You should therefore not rely on this being true.

The read/write (data) area may contain code as programs are sometimes self-modifying. Similarly, the read-only (code) area may contain read-only data (eg string literals in C; floating-point constants; etc).

These symbols can be imported as relocatable addresses by assembly language routines that might need them.

# INDEX